



CS1Ah Revision

Computer Science
School of Informatics
The University of Edinburgh



Java revision

CS1Ah had 10 lectures on Java, which were:

- **Introduction** — HelloWorld and SumTwo
- **Types** — primitives, wrappers, BigInteger
- **Expressions** — operators, overflow,
- **Control** — assignments, **if-then-else**, blocks, **while**-iteration
- **Classes and objects** — `Student`, **private**, `==` and `.equals`
- **Inheritance** — class hierarchy, `Object`, **instanceof**
- **Arrays** — **for** bounded iteration
- **Streams and Exceptions** — input and output, **try-catch**
- **Parameter passing** — call-by-value, reference parameter
- **Collection framework** — interfaces, implementations, algorithms

We will revise all but the last of these lectures in the following slides. The slides are a summary and *do not cover everything that can come up in the exam.*

Non-Java revision

Apart from Java programming, CS1Ah included lectures on several other topics. We will revise some material on

- **Finite state machines and regular expressions**
- **Logical Expressions**
- **Data structures** — lists, sets, stacks, queues, trees
- **Algorithms** — sorting

Other topics not covered in these slides are

- **Introduction to software engineering, object-oriented design**
- **Case studies**

Remember that *all of the material taught in lectures is examinable.*

Hello Java

```
// HelloWorld.java: print a message to the world
```

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

This program shows several basic constituents of a program. You should be able to identify the following.

- Comments, whitespace.
- Class definition.
- Method definition: header and body.
- Method invocation.
- String literal.
- Statement.

A program with input

```
class SumTwo {  
    public static void main(String[] args) {  
        int x = Integer.parseInt(args[0]);  
        int y = Integer.parseInt(args[1]);  
        System.out.print("The sum is: ");  
        System.out.println(x+y);  
    }  
}
```

This program shows some more constituents of a program. You should be able to identify the following.

- Types `void` and `String[]`.
- Modifiers `public`, `static`.
- Variable declaration, initialization.
- `Integer.parseInt` method.

Java types

Java provides the *primitive* types of boolean, char, byte, short, int, long, float and double.

For example, it is never meaningful to add an integer to a boolean, or to subtract two boolean values. So any Java program which contains such an expression is rejected.

Integer arithmetic can be performed more efficiently than floating-point arithmetic so by using types to separate out floating-point values from integer values we can reduce the execution time of programs.

Type	Size	Minimum value	Maximum value
<i>boolean</i>	1 bit	The boolean type is not ordered	
<i>char</i>	16 bits	\u0000	\uFFFF
<i>byte</i>	8 bits	Byte.MIN_VALUE (-2^7)	Byte.MAX_VALUE ($2^7 - 1$)
<i>short</i>	16 bits	Short.MIN_VALUE (-2^{15})	Short.MAX_VALUE ($2^{15} - 1$)
<i>int</i>	32 bits	Integer.MIN_VALUE (-2^{31})	Integer.MAX_VALUE ($2^{31} - 1$)
<i>long</i>	64 bits	Long.MIN_VALUE (-2^{63})	Long.MAX_VALUE ($2^{63} - 1$)
<i>float</i>	32 bits	Float.MIN_VALUE	Float.MAX_VALUE
<i>double</i>	64 bits	Double.MIN_VALUE	Double.MAX_VALUE

Converting between types and objects

Converting a literal to an object	Converting an object to a value
<pre>Byte B = new Byte((byte)32);</pre>	<pre>byte b = B.byteValue();</pre>
<pre>Short S = new Short((short)512);</pre>	<pre>short s = S.shortValue();</pre>
<pre>Integer I = new Integer(1024);</pre>	<pre>int i = I.intValue();</pre>
<pre>Long L = new Long(2048L);</pre>	<pre>long l = L.longValue();</pre>

Extending the primitive types

```
import java.math.BigInteger;
class Powers {
    public static void main(String[] args) {
        final BigInteger two = new BigInteger("2");
        BigInteger I = new BigInteger("1");
        while (true) {
            I = I.multiply(two);
            System.out.println(I);
        }
    }
}
```

Arithmetic operators

Symbol	Example	Name
-	- a	unary minus
*	a * b	multiplication
/	a / b	division
%	a % b	remainder (mod)
+	a + b	addition
-	a - b	subtraction

In each section of the table, operators have equal precedence; sections are ordered from higher to lower precedence.

All binary arithmetic operators are left associative, i.e. evaluated from left to right. For example, $a/b*c$ means $(a/b)*c$.

How should overflow be handled?

Java does not view overflow and underflow as fatal errors which should stop the execution of a program.

One justification for this is that it is possible for numbers to overflow and then later to underflow back to the correct answer.

Expression	Result
<code>Integer.MAX_VALUE</code>	2147483647
<code>Integer.MAX_VALUE + 1</code>	-2147483648
<code>(Integer.MAX_VALUE + 1) - 1</code>	2147483647

Prefix and postfix increment and decrement

```
x = 190;           // sets x to 190
System.out.println(x++); // prints 190
System.out.println(x);   // prints 191
System.out.println(x--); // prints 191
System.out.println(x);   // prints 190
```

```
x = 190;           // sets x to 190
System.out.println(++x); // prints 191
System.out.println(x);   // prints 191
System.out.println(--x); // prints 190
System.out.println(x);   // prints 190
```

Boolean expressions

Operator	Example	Description
==	1 == 1	equal to
!=	1 != 2	not equal to
<	1 < 2	less than
>	2 > 1	greater than
<=	1 <= 1	less than or equal to
>=	2 >= 1	greater than or equal to

Note that == denotes the equality test whereas = denotes assignment.

Assignments

Statement	Equivalent	Effect
<code>x *= 2;</code>	<code>x = x * 2;</code>	multiplies x by two
<code>x /= 2;</code>	<code>x = x / 2;</code>	divides x by two
<code>x %= 2;</code>	<code>x = x % 2;</code>	sets x to x remainder two
<code>x += 2;</code>	<code>x = x + 2;</code>	adds two to x
<code>x -= 2;</code>	<code>x = x - 2;</code>	subtracts two from x

The equals sign is an *assignment operator* in Java.

The operators `*=`, `/=`, `%=`, `+=` and `-=` are also assignment operators, and there are still others.

These operators combine the evaluation of an expression and the execution of an assignment into a single statement.

Conditional statements: examples

```
if (x == 0)
    System.out.println("zero");
```

```
if (x == 0)
    System.out.println("zero");
```

```
if (x != 0)
    System.out.println("non-zero");
```

```
if (x == 0)
    System.out.println("zero");
else
    System.out.println("non-zero");
```

Statement blocks

Statement	Effect
<pre>if (x < 0) { System.out.println("negative"); x *= -1; }</pre>	Changes the sign of x and prints negative if x is negative.
<pre>if (x < 0) System.out.println("negative"); x *= -1;</pre>	Prints negative if x is negative. Changes the sign of x whether it was negative or not.

Iteration

To express *repetitive* execution of statements we use a **while** statement.

```
while (x > 0) {  
    System.out.println ("decreasing");  
    x--;  
}  
System.out.println ("finished");
```

The two statements which come between the open brace and the close brace are called the *body* of the loop. The body is repeatedly executed while the loop *condition* is true.

There is a possibility that the loop body will never be executed at all.

Objects

The Java programming language uses *objects* to store related items of data.

Objects belong to *classes* which define the information which an object *instance* of that class contains.

The relationship of an object to its class can then be likened to the relationship of a value to its type although there is a subtle difference. An object instance can be `null`, whereas a value cannot.

The value of an `int` variable must always be an integer whereas a `String` object for example will either be a string or `null`.

Every string, even the empty string, is non-null and the same is true for every class of object, the *constructed* object instances are non-null.

A simple class for student data

```
class Student4 {  
    public String surname;  
    public String forename;  
    public String matricno;  
}
```

In programming languages which do not have objects such a definition is called a *record* or a *structure* (sometimes '*struct*').

An object is *first-class* in the sense that it can be passed to a method or returned as its result.

Constructors

```
public Student4(String surname,  
               String forename, String matricno) {  
    this.surname = surname;  
    this.forename = forename;  
    if (matricno.length() != 7) {  
        this.matricno = null;  
    } else {  
        this.matricno = matricno;  
    }  
}
```

Private fields and accessor methods

```
private String surname;  
private String forename;  
private String matricno;
```

```
public String getSurname() {  
    return surname;  
}
```

```
public String getForename() {  
    return forename;  
}
```

```
public String getMatricno() {  
    return matricno;  
}
```

When are objects equal?

Consider the following lines from a Java program.

```
Student4 Jo      = new Student4("Smith", "Jo", 0112345);  
Student4 Josie   = new Student4("Smith", "Jo", 0112345);
```

Jo and Josie are *different* objects with equal contents. Java provides two ways of testing objects for these two notions of equality.

```
Jo == Josie           returns false  
Jo.equals(Josie)     returns true
```

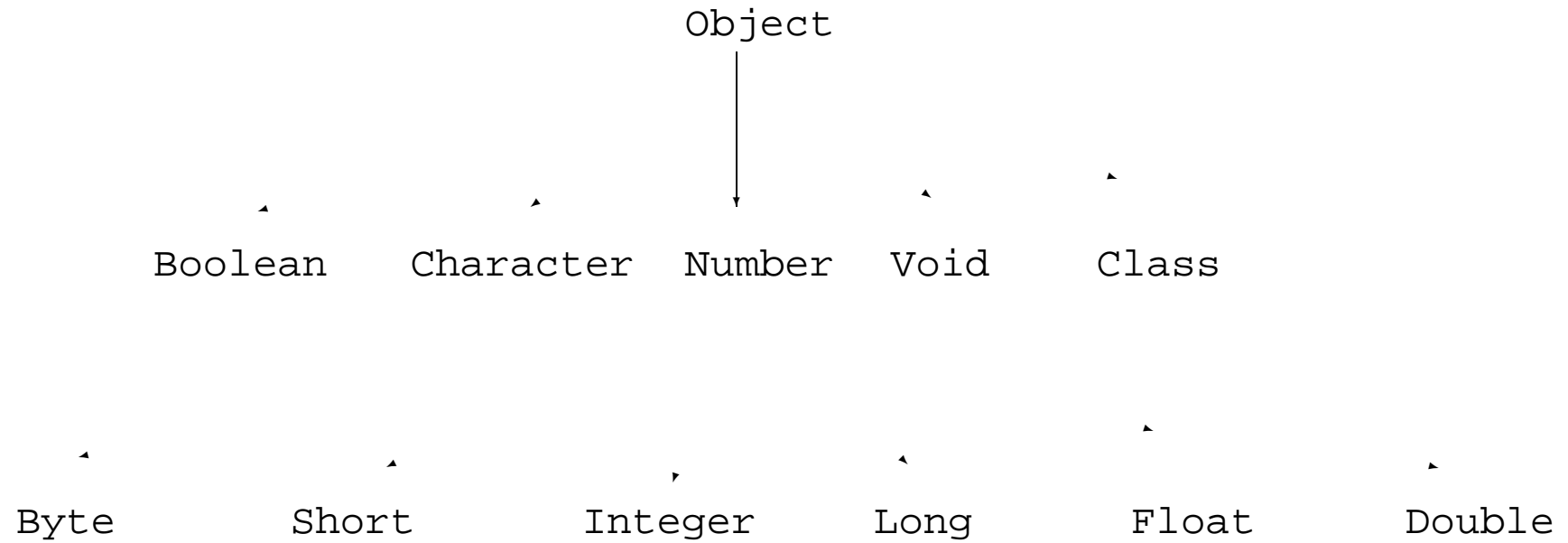
Objects and assignment

By assigning one object to another we will make them identical. If we execute this Java assignment `Jo = Josie;` and then repeat the two tests for the two notions of equality we find this:

```
Jo == Josie           returns true
Jo.equals(Josie)     returns true
```

If two objects are identical then they must have identical contents so once the `==` operator has returned true then we know that the `equals()` method will return true also. In the context of its use with objects, the `==` operator is sometimes called *pointer equality*.

The Class Hierarchy



Extending the Hierarchy by Subclassing

A class declaration can include the phrase **extends** *c*

```
class CompSciStudent extends Student {  
    public String username;  
    public int diskquota;  
}
```

As well as new fields, a subclass may add new methods

Or a subclass may *replace* methods of its superclass with its own versions, which is called *method overriding*.

Polymorphism

Vectors are a *polymorphic* data structure (“polymorphic” = “having many forms”). This means that the same operations can be used for vectors of integer objects as vectors of strings or vectors of dates.

In contrast, many programming languages only support only the definition of *monomorphic* data structures (“monomorphic” = “having only one form”). In a monomorphic programming language, separate implementations of the vector operations would be needed for vectors of integer objects, vectors of strings, or vectors of dates.

The use of a single definition which is re-usable at different types (i.e. polymorphic) saves programmer effort and thereby reduces the chance of error.

Processing objects by type

When we retrieve objects from a collection we need to restore the class which they had before they were coerced to `Object`.

Casting up the class hierarchy, an operation which is sometimes called *widening*, is always safe (every `Integer` is a `Number` ...).

However, casting down the class hierarchy, an operation which is sometimes called *narrowing*, is not always safe (not every `Number` is an `Integer` ...).

If we get this down-casting wrong the operation will fail at run-time with a `ClassCastException`. We can investigate the class of an object at run-time by the use of the **`instanceof`** relation.

Using arrays

An array is a structure of related variables, which can be *indexed* to access particular elements. For example, here is an array `p` that stores the first five prime numbers:

	<code>p[0]</code>	<code>p[1]</code>	<code>p[2]</code>	<code>p[3]</code>	<code>p[4]</code>
<code>p</code>	2	3	5	7	11

Arrays are indexed from zero. We use `p[0]` to access the zeroth entry in `p`, which has the value 2. We would use `p[4]` to access the last entry in the array, which has the value 11.

A FOR loop has four parts

The initialisation: This typically defines a *loop control variable* whose value changes on each iteration of the loop. The initialisation statement in the header of a **for** loop is only executed once.

The condition: This typically uses the value of the loop control variable to determine whether or not the work of the loop is complete.

The loop body: The body is executed repeatedly while the condition evaluates to true. If the loop condition is false initially then the loop body will not be executed at all.

The update: After each execution of the body of the **for** loop the loop control variable is updated.

Sieve Prime Numbers

```
class Sieve {
    public static void main( String[] args) {
        final int N = 1000;
        boolean[] sieved = new boolean[N];
        for ( int i = 2 ; i < N ; i++) {
            if (sieved[i]) continue;
            System.out.println(i);
            for ( int j = 2 ; i * j < N ; j++) {
                sieved[i * j] = true;
            }
        }
    }
}
```

Multi-dimensional arrays

An ordinary array stores a single row of data. Sometimes we want to organise data elements in tables, or tables of tables...

We can use a *multi-dimensional array* for this:

	q[0][0]	q[0][1]	q[0][2]	q[0][3]	q[0][4]
q[0]	2	3	5	7	11
	q[1][0]	q[1][1]	q[1][2]	q[1][3]	q[1][4]
q[1]	4	6	10	14	22

Java treats multi-dimensional arrays as arrays of arrays.

Multi-dimensional arrays do not need to be rectangular.

Recursion and iteration

```
public static int factorial(int n) {  
    if (n==1) return 1;  
    else return n * factorial (n-1);  
}
```

```
public static int factorial(int n) {  
    int ans = 1;  
    while (n != 1) {  
        ans *= n;  
        n--;  
    }  
    return ans;  
}
```

A serializable student record

```
class Student5 implements java.io.Serializable {
    private String surname;
    private String forename;
    private String matricno;

    // the following method is the constructor
    public Student5( String surname,
                    String forename, String matricno) {
        this.surname = surname;
        this.forename = forename;
        if (matricno.length() != 7) {
            this.matricno = null;
        } else {
            this.matricno = matricno;
        }
    }
    // and other methods...
}
```

Stream output

```
import java.io.*;
class StudentOutput {
    public static void main(String[] args) {
        try {
            // Create an output stream of student records
            String name = "Students.dat";
            FileOutputStream file = new FileOutputStream(name);
            ObjectOutputStream out =
                new ObjectOutputStream(file);

            Student5 JoSmith = new Student5
                ("Smith", "Jo", "0012345");
            Student5 MikeSmith = new Student5
                ("Smith", "Mike", "0012346");

            out.writeObject(JoSmith);
            out.writeObject(MikeSmith);

            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Stream input

```
while (true) {
    try {
        Student5 s = (Student5) in.readObject();
        System.out.println(s);
    } catch (EOFException e) {
        System.out.println("End of report");
        break;
    } catch (ClassCastException e) {
        System.out.println("Invalid record");
    }
}
```

Exceptions

Exceptional situations are handled in Java as *exceptions*.

Knowing that there is the possibility of failure, we *try* to execute a block of statements and we *catch* an exception if it occurs, executing another block of statements in that case (**try** and **catch** are reserved words).

Different exceptions are *thrown* to indicate different kinds of errors.

If exceptions do not occur then the **try** block completes and the statements in the **catch** block are not executed.

Call by value

Java makes *copies* of the actual parameters supplied to a method, which take the place of the formal parameters during the method invocation. This is called *call-by-value*.

```
class CallByValue {  
    static void assignString(String st) {  
        st = "The string in the method";  
    }  
    public static void main(String[] args) {  
        String st = "The default initial string";  
        assignString(st);  
        System.out.println(st);  
    }  
}
```

Passing references

To update something inside a method via a parameter, we need to *pass a reference* to it. Java doesn't allow arbitrary manipulation of references, but objects and arrays are always references.

```
class UseStringRef {  
    static class StringRef { public String st; }  
    static void assignStringRef(StringRef r) {  
        r.st = "The string in the method";  
    }  
    public static void main(String[] args) {  
        StringRef r = new StringRef();  
        r.st = "The default initial string";  
        assignStringRef(r);  
        System.out.println(r.st);  
    }  
}
```

FSM - Formal Definition

- An **input alphabet**, Σ (“sigma”), which is a set of symbols, one for each distinct possible input (*e.g.* buttons, coin slots, ...)
 - An **output alphabet**, Λ (“lambda”), which is another set of symbols, one for each distinct possible output (*e.g.* lights, can dispensed, ...)
 - Some named **states**, each capturing some state of the system (*e.g.* “coin inserted but no can dispensed”), drawn as circles, one indicated as the **start state** with an arrow
 - A collection of **transitions**, each an arrow between two states, labelled with the input which activates it and the output which results (or ϵ (“epsilon”) indicating no input or output respectively)
-

Acceptors

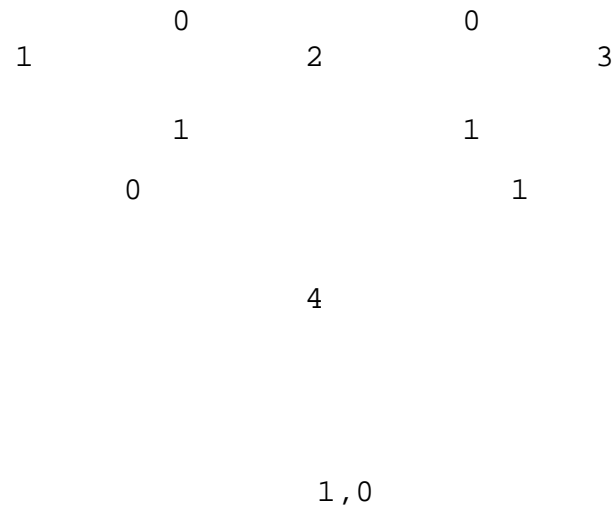
Acceptors are FSMs with

- empty output alphabet (all outputs are ϵ)
- some states marked as “accepting” (double circle)

Input sequence is **accepted** if it can generate a trace from the start state to an accepting state.

The **language accepted by an (acceptor) FSM is the set of all input sequences which it accepts.**

Acceptors



Accepts strings of 0s and 1s in which the number of 0s so far never exceeds the number of 1s so far by more than one and *vice versa*

Propositional logic

The components of a logical expression are:

logical constants, t and f (denoting the *truth values* true and false),

propositional variables such as P , Q and others as needed, and

the **logical operators** or *connectives*:

- \neg ('not' or 'negation')
- \wedge ('and' or 'conjunction')
- \vee ('or' or 'disjunction')
- \Rightarrow ('implies' or 'implication')
- \Leftrightarrow ('if and only if', 'iff', 'equivalence' or 'bi-implication')

Logical expressions

A logical expression can be defined thus:

The constants t and f are logical expressions.

The propositional variables are logical expressions.

If P is a logical expression then so are (P) and $\neg P$.

If P and Q are logical expressions so are $P \wedge Q$, $P \vee Q$, $P \Rightarrow Q$ and $P \Leftrightarrow Q$.

Notice that this is a *recursive* definition. It defines an *algebra* for logical expressions which is often called *Boolean algebra* after its inventor, the logician George Boole.

Truth tables

The meaning of the logical operators can be presented in a *truth table* which inspects a logical expression by examining all of the possible assignments of truth values to the propositional variables in the expression.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
f	f	t	f	f	t	t
f	t	t	f	t	t	f
t	f	f	f	t	f	f
t	t	f	t	t	t	t

A **tautology** is a logical expression whose value is always t for all of the possible assignments of truth values to its propositional variables. An example of a tautology is $P \vee \neg P$. We can confirm that this is a tautology by completing the truth table for this expression.

Laws of equivalence

1. $P \Leftrightarrow P$
 2. $(P \Leftrightarrow Q) \Leftrightarrow (Q \Leftrightarrow P)$
 3. $((P \Leftrightarrow Q) \wedge (Q \Leftrightarrow R)) \Rightarrow (P \Leftrightarrow R)$
 4. $(P \Leftrightarrow Q) \Leftrightarrow (\neg P \Leftrightarrow \neg Q)$
-

Laws analogous to arithmetic

5. $(P \wedge Q) \Leftrightarrow (Q \wedge P)$

10. $(P \wedge t) \Leftrightarrow P$

6. $(P \wedge (Q \wedge R)) \Leftrightarrow ((P \wedge Q) \wedge R)$

11. $(P \vee f) \Leftrightarrow P$

7. $(P \vee Q) \Leftrightarrow (Q \vee P)$

12. $(P \wedge f) \Leftrightarrow f$

8. $(P \vee (Q \vee R)) \Leftrightarrow ((P \vee Q) \vee R)$

13. $(\neg \neg P) \Leftrightarrow P$

9. $(P \wedge (Q \vee R)) \Leftrightarrow ((P \wedge Q) \vee (P \wedge R))$

Differences from arithmetic

$$14. (P \vee (Q \wedge R)) \Leftrightarrow ((P \vee Q) \wedge (P \vee R))$$

$$15. (t \vee P) \Leftrightarrow t$$

$$16. (P \wedge P) \Leftrightarrow P$$

$$17. (P \vee P) \Leftrightarrow P$$

$$18a. (P \vee (P \wedge Q)) \Leftrightarrow P$$

$$18b. (P \wedge (P \vee Q)) \Leftrightarrow P$$

$$19a. (P \wedge (\neg P \vee Q)) \Leftrightarrow (P \wedge Q)$$

$$19b. (P \vee (\neg P \wedge Q)) \Leftrightarrow (P \vee Q)$$

De Morgan's laws and implication

$$20a. \quad \neg(P \wedge Q) \Leftrightarrow (\neg P \vee \neg Q)$$

$$20b. \quad \neg(P \vee Q) \Leftrightarrow (\neg P \wedge \neg Q)$$

$$21. \quad ((P \Rightarrow Q) \wedge (Q \Rightarrow P)) \Leftrightarrow (P \Leftrightarrow Q)$$

$$22. \quad (P \Leftrightarrow Q) \Rightarrow (P \Rightarrow Q)$$

$$23. \quad ((P \Rightarrow Q) \wedge (Q \Rightarrow R)) \Rightarrow (P \Rightarrow R)$$

$$24. \quad (P \Rightarrow Q) \Leftrightarrow (\neg P \vee Q)$$

Linked lists in Java

The `java.util` package provides an implementation of doubly linked lists in its `LinkedList` class.

`addFirst()`, `addLast()` These methods allow us to add an object at the beginning or the end of the list.

`getFirst()`, `getLast()` These methods allow us to inspect the object at the beginning or the end of the list.

`removeFirst()`, `removeLast()` These methods allow us to delete an object at the beginning or the end of the list.

The `add()`, `get()` and `remove()` methods are parameterised by an integer allowing us to specify other positions in the list.

There is a `set()` method which allows us to set the contents of the cell at a particular position. As with arrays, numbering is from zero.

Sets and arrays

Lists allow us to assemble collections of objects which are stored in some order. For some applications all we need is a collection of objects so that we may ask if the collection contains a particular element or not.

A collection where order is not important is called a *set*.

Like arrays, sets provide efficient access to every element. Sets differ from arrays in that access is provided by *content*, not by *position*.

The type of questions which we pose with a set are whether or not an element is included. With lists and arrays we could ask where an element appears or how many times an element appears but these questions are not meaningful for sets.

Implementations of sets

The `java.util` package provides two implementations of sets, `HashSet` and `TreeSet`. These provide the following methods.

`add()`, `addAll()` These methods allow us to add one or several objects to the set.

`contains()`, `containsAll()` These methods allow us to test whether one or several objects are in the set.

`remove()`, `removeAll()` These methods allow us to delete one or several objects.

A `TreeSet` maintains the elements of the set in sorted order because this makes adding and removing elements and testing for membership more efficient.

A `HashSet` also keeps the elements in order (but a different order).

Trees

A **tree** consists of a collection of **nodes** connected by directed **edges**.

Terminology you should know: *parent*, *child*, *root*, *leaf*, *subtree*, *path*, *height*.

Two definitions of the set of all trees: (1) cutting down from the set of *graphs*, and (2) building up inductively from a single node, or a node connecting some positive number of subtrees.

Example trees: *expression trees* (aka syntax trees: nodes contain operations, leaves contain variables/values); *tries* (edges are labelled, path codes element).

Variants: *n-ary trees* and *ordered trees*.

Binary trees

Special case of ordered n -ary tree with $n = 2$, but with fixed *left* and *right* positions.

Inductive definition: build up from empty tree or a node with two subtrees. Thus every node has exactly two subtrees.

Traversing (binary) trees: **pre-order**, **in-order** and **post-order**. Also generalised **Euler tour**.

Representing trees in Java: array based representations (good for almost full trees), dynamically managed memory representation: nodes are objects.

The binary search tree property

A binary search tree relies on an ordering on data elements

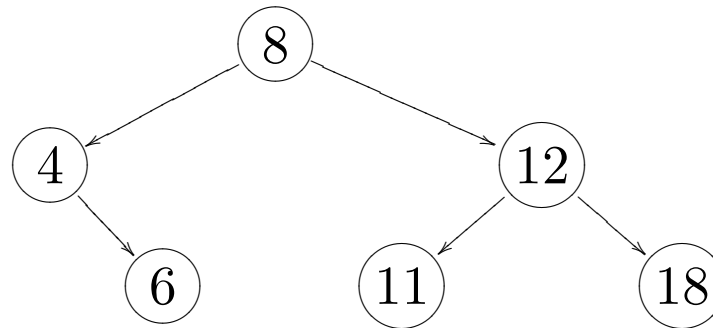
The ordering helps enforce the *binary search tree property*:

For each node n , the

- labels in n 's **left** subtree T_L are all **less than** the label of n , and
- labels in n 's **right** subtree T_R are all **greater than** the label of n .

In particular, this means that T_L and T_R also satisfy the property.

Example search tree



An in-order traversal of the tree will visit the nodes in ascending order.

Equivalent trees: remember that the search tree property doesn't restrict the shape of the tree. (What does equivalent mean?)

Searching in the tree

Here is an algorithm to search the tree T for an item d , returning a boolean value which is `true` if d appears in T .

- If T is empty, return `false`
- Otherwise, compare d with the label d_r of the root of T
- If it is equal, return `true`.
- If $d < d_r$, recursively search the left subtree of T
- If $d > d_r$, recursively search the right subtree of T

In fact, several basic operations will use this same recursion pattern.

The SearchTree class

```
// File: SearchTree.java
class SearchTree implements MyCollection {
    static class Node {
        Node      left;
        Comparable data;
        Node      right;

        public Node(Comparable d) {
            data = d;    // handy constructor
        }
    }
    private Node root;

    // operations follow here...
}
```

The contains operation — public method

Our implementation follows a general pattern: we use a short public instance method to invoke a private *static* method on the `root`. The static method uses recursion.

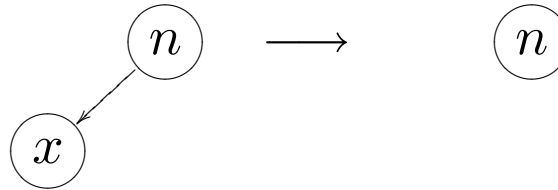
Here is the public method for the **contains** operation:

```
// Is item in the tree?  
public boolean contains(Comparable item) {  
    return contains(root, item);  
}
```

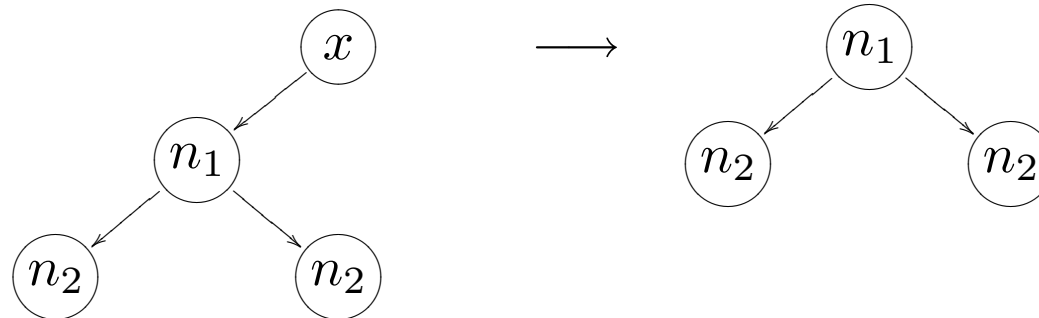
The remove operation

The remove operation is a bit more complicated. We need to think about the possible cases for removing a node:

- Removing a leaf from the tree is easy...



- and so is removing a node which has a single subtree.



- But what about removing an internal node with two subtrees?

Removing an internal node

To remove an internal node, we could *merge* the two subtrees.

Alternatively, we can “cheat” by deleting a leaf node instead, and moving its label to the internal node.

We must choose a leaf node for which this is possible. It can either be the **greatest element in the left** subtree, or the **least element in the right** subtree.

We will choose the latter here. See one of the exercises in Deitel & Deitel for an implementation which makes the other choice.

Complexity Analysis

We can analyse the running time (or memory consumption) of algorithms in the best, worst, and average case.

Worst case: $T(n)$ = maximum running time over all inputs of size n .

Best case: $T(n)$ = minimum running time over all inputs of size n .

Average case: $T(n)$ = sum of all running times over all inputs of size n , divided by the number of inputs of size n .

O-notation

Gives reasonable information about the running-time of an algorithm (more precisely, about how fast it grows with n), and is reasonably easy to estimate.

We say $T(n)$ 'is' $O(f(n))$ if there is a constant c and a number n_0 such that $T(n) \leq c \cdot f(n)$ for every $n \geq n_0$.

$O(f(n))$ says that $f(n)$ is an upper bound on the growth-rate (up to constant factors). $\Theta(f(n))$ says that $f(n)$ is the growth rate (up to constant factors).

Complexity analysis of tree operations

Each operation is based on a recursion which follows a path down the tree, comparing labels. The number of recursive calls is equal to the length of the path followed; each method takes $O(1)$ plus the time for the recursive call; the longest path is equal to the height of the tree, h . In general, any operation is $O(h)$. How does h relate to the size of the tree?

Worst case The worst case tree degenerates to a linear single subtree pattern, so the height $h = n - 1$ and our operations are $O(n)$.

Best case If the tree is more *balanced*, or even *full*, then the height is $\log_2(n)$ rounded up. So in the best case, our operations are $O(\log n)$.

Average case On average, trees tend to be more balanced than linear; it can be shown that the average case also leads to $O(\log n)$ complexity.

Sorting

Sorting algorithms can be in-place or not, online or offline, comparison-based or not, etc. The best sorting algorithm depends on the application.

Selection and insertion sort are comparison-based algorithms with $O(n^2)$ worst case running time. (Selection sort has also $O(n^2)$ best case running time, while insertion sort has $O(n)$ (sorted sequence).)

Mergesort is a comparison-based algorithms with $O(n \log n)$ worst case (and best case) running time.

Quicksort is a comparison-based algorithm with $O(n^2)$ worst case running time (sequence sorted in inverse order) but $O(n \log n)$ average running time.

Bucketsort is not comparison-based. It sorts n numbers numbers in the range $0..m - 1$ in $O(n + m)$ worst case (and best case) time