The Open University

Undergraduate ICT and Computing: Level 2

M257 Putting Java to work

M257 Exam Handbook

Important: This Exam Handbook may be taken into the Examination but it must not contain any annotations or any other additions except for any official Errata that appear on the M257 website.

The materials are drawn chiefly from the course units and Java APIs. Although this handbook includes material not discussed in M257, because we may wish to refer to such material in an exam, this does not imply a need to study details outside of the course materials.

This booklet is not designed to be read from cover to cover; rather you should use its table of contents to find the documentation for a particular method, class or interface. The information presented has generally been abbreviated.

Methods and constructors are public unless otherwise indicated.

Contents

ISBN 978 07492 1996 3
Printed in the United Kingdom
2.1

1 Some Java syntax

1.0 Keywords

`abstract`	`case`	`default`	`final`	`if`	`long`	`return`	`this`
`boolean`	`catch`	`do`	`finally`	`implements`	`new`	`short`	`throw`
`break`	`char`	`double`	`float`	`instanceof`	`package`	`static`	`throws`
`byte`	`class`	`else`	`for`	`int`	`private`	`super`	`try`
	`continue`	`enum`		`interface`	`protected`	`switch`	`void`
		`extends`			`public`	`synchronized`	`while`

1.1 Operators

Arithmetic and assignment operators

Symbol	Operator	Written as	Meaning
`*`	multiplication	`a * b`	`a` times `b`
`/`	division	`a / b`	`a` divided by `b`
`%`	remainder	`a % b`	the remainder of (`a` divided by `b`)
`+`	addition	`a + b`	`a` added to `b`
`-`	subtraction	`a - b`	`a` minus `b`
`++`	postfix increment	`x++`	increments `x` by 1 and returns the old value
`++`	prefix increment	`++x`	increments `x` by 1 and returns the new value
`--`	postfix decrement	`x--`	decrements `x` by 1 and returns the old value
`--`	prefix decrement	`--x`	decrements `x` by 1 and returns the new value
`=`	assign	`a = b`	put the value of `b` into `a`
`+=`	add and assign	`a += b`	put the value of `a + b` into `a`
`-=`	subtract and assign	`a -= b`	put the value of `a – b` into `a`
`*=`	multiply and assign	`a *= b`	put the value of `a * b` into `a`
`/=`	divide and assign	`a /= b`	put the value of `a / b` into `a`
`%=`	remainder and assign	`a %= b`	put the value of `a % b` into `a`

Logical and relational operators

Symbol	Operator	Written as	Meaning
`&&`	Logical and	`a && b`	returns true if both `a` and `b` are true, otherwise false
`||`	Logical or	`a || b`	returns false if both `a` and `b` are false, otherwise true
`!`	Logical negation	`!a`	returns false if `a` is true, returns true if `a` is false
`==`	equal to	`x == y`	true if `x` equals `y`, otherwise false
`>`	greater than	`x > y`	true if `x` is greater than `y`, otherwise false
`<`	less than	`x < y`	true if `x` is less than `y`, otherwise false
`>=`	greater than or equal to	`x >= y`	true if `x` is greater than or equal to `y`, otherwise false
`<=`	less than or equal to	`x <= y`	true if `x` is less than or equal to `y`, otherwise false
`!=`	not equal to	`x != y`	true if `x` is not equal to `y`, otherwise false

1.2 Iteration

Keyword	Meaning	Example of use

`for`	A looping statement indicating a starting value of a control variable, a condition to evaluate that determines when the loop should end, and an adjustment to the control variable, together with statements to be performed while the loop condition holds.	`for (int j = 1; j < 10; j++)`
		`{`
		`  System.out.println(j + ")");`
		`}`

	A variant form of for-loop called for-each iterates over the objects contained in a collection implementing `Iterable`, or an array, and extracts each of them. If the collection guarantees ordering, that order is used.	`for (Pupil p : pupilCollection)`
		`{`
		`  if (p.getAge() > 10)`
		`  {`
		`  System.out.println(p);`
		`  }`
		`}`

`while`	A looping statement, the body of which will be repeatedly executed as long as some condition continues to evaluate to `true`.	`while (countdown != 0)`
		`{`
		`  System.out.println(countdown);`
		`  countdown--;`
		`}`

	A variant form of while begins with the keyword `do`. In this case, the condition is only checked after the first execution of the body.	`do`
		`{`
		`  System.out.println(countdown);`
		`  countdown--;`
		`}`
		`while(countdown > 1);`

1.3 Selection

Keyword	Meaning	Example of use

`if`	A selection statement, allowing the flow of program control to be changed. A logical condition is evaluated and a block of code executed only if the condition evaluates to `true`.	`if (day == 0)`
		`{`
		`  System.out.println("Sunday");`
		`}`

	Optionally the keyword `else` can be used to specify code to be executed if the condition evaluates to `false`.	`if (num % 2 == 0)`
		`{`
		`  System.out.println("even");`
		`}`
		`else`
		`{`
		`  System.out.println("odd");`
		`}`

`switch`	This keyword is used for a selection statement allowing choice of one of several paths of program control depending on the value of an argument (here `val`) of primitive, `enum` or wrapper type.	`switch (val)`
		`{`
		`  case 'a':`
		`  {`
		`  videoId = 3;`
		`  break;`
		`  }`
		`  case 'b':`
		`  {`
		`  videoId = 19;`
		`  break;`
		`  }`
		`  default:`
		`  {`
		`  System.out.println("42");`
		`  break;`
		`  }`
		`}`

1.4 Arrays

Aspects	Example of use

A linear form of storage, indexed by an `int` beginning from `0`. An array can be declared using the type it stores, followed by square brackets.	`int[] ia; //array of ints`
	`String[] sa; //array of String refs`

Arrays are instantiated using the keyword `new` and then the type, with a size argument.	`ia = new int[4]; //room for 4 ints`
Default values are used to initialize an array if no explicit initialization is performed. You can also use an array initializer when creating an array.	`char[] myChars = {'a', 'c', 'x'};`
Array contents are accessed using an `int` index.	`char f = myChars[0];`
An array's length can be found using its `length` instance data.	`int len = ia.length;`

1.5 Exceptions

Keyword	Meaning	Example of use

`try`	Introduces a block of code in which an exception can occur.	`try`
		`{`
		`  //something`
		`}`

`catch`	Follows a try block and introduces code to handle an exception of the specified type.	`catch(Exception ex)`
		`{`
		`  System.out.println("Caught exception ex.getMessage());`
		`}`

`finally`	Introduces a block of code that will always be performed, after a try-catch statement.	`finally`
		`{`
		`  //always do something`
		`}`

`throws`	Used to Indicate that one or more exceptions (separated by commas) may be thrown by a method or constructor.	`throws Exception`
`throw`	Used to throw an exception.	`throw new Exception();`

2 Java API excerpts

2.0 Errors and exceptions

The following table includes a number of common exception types.

Class	Meaning
`ArithemeticException` `extends RunTimeException`	Thrown when an illegal arithmetic condition occurs for integral types.
`ArrayIndexOutOfBoundsException` `extends RunTimeException`	Thrown by an attempt to access an element in an array which is outside its declared boundaries.
`EOFException` `extends IOException`	Thrown on attempting to read past the end of a file or stream.
`Exception` `extends Throwable`	Top-level class of checked exceptions.
`FileNotFoundException` `extends IOException`	Thrown on attempting to open or write to a file and the file is not found.
`IOException` `extends Exception`	General class of exceptions due to failed or interrupted input and output operations.
`MalformedURLException` `extends IOException`	Thrown on constructing an incorrectly formed URL.
`NullPointerException` `extends RunTimeException`	Thrown on an attempt to access an object from a null reference.
`RunTimeException` `extends Exception`	Top-level class of exceptions that may be thrown by the virtual machine at runtime (unchecked).
`Throwable` `extends Object`	Top-level class of all errors and exceptions (unchecked and checked).
`UnsupportedOperationException` `extends RunTimeException`	Thrown to indicate that the requested operation is not supported; for example, if an optional interface operation is not supported.
`Error` `extends Throwable`	Top-level class of exceptions that occur when some internal Java error has happened – for example, the Java system has run out of memory (unchecked).

2.1 Some common methods of the `Thread` class

The Java Virtual Machine allows an application to have multiple threads of execution running concurrently. Threads are instances of the `Thread` class.

Sample methods	**Meaning**
`String getName()`	Returns this thread's name.
`int getPriority()`	Returns this thread's priority.
`void join()`	Waits for this thread to die.
`void run()`	If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's `run` method is called; otherwise, this method does nothing and returns.
`void setName(String name)`	Changes the name of this thread to be equal to the argument `name`.
`void setPriority(int priority)`	Changes the priority of this thread.
`static void sleep(long millis)`	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
`void start()`	Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.
`String toString()`	Returns a string representation of this thread, including the thread's name, priority, and thread group.
`static void yield()`	Causes the currently executing thread object to temporarily pause and allow other threads to execute.

2.2 Some common methods of the `String` class

Sample Methods	Meaning
`char charAt(int i)`	Finds and returns the character at position `i`.
`boolean equals(Object o)`	Compares this string to the specified object.
`int indexOf(int ch)`	Returns the index within this string of the first occurrence of the specified character or `-1` if not found.
`int indexOf(String str)`	Searches for a particular string `str` within the destination object and returns the position of the first index of `str` within the string. The search starts at the front of the string.
`int lastIndexOf(String str)`	This is the same as `indexOf` but searches from the back of the destination string object.
`int length()`	Returns the length of the string.
`String substring(int beginIndex,int endIndex)`	This returns a substring starting at position `beginIndex` and ends at position `endIndex – 1` of the destination string object.
`String toLowerCase()`	Returns the characters in this `String` in lower case using the rules of the default locale.
`String toUpperCase()`	Returns the characters in this `String` in upper case using the rules of the default locale.
`String valueOf(x)`	Returns a string representation of the argument x, which may be of any primitive type, or of type `Object`.

2.3 Some important interfaces

In some cases* we have listed only the more commonly required methods defined by the interface. The list of classes that implement the interfaces are not intended to be complete.

Interface	Sample methods	Implemented by

`Collection<E>`
`extends Iterable<E>`
*
The root interface in the collection hierarchy representing a group of objects, known as its elements.
	`boolean add(E o)`
	`boolean contains(Object o)`
	`boolean isEmpty()`
	`Iterator<E> iterator()`
	`boolean remove(Object o)`
	`int size()`
	`Object[] toArray()`
		The JDK provides implement-ations only of more specific subinterfaces like `Set` and `List`.

`Comparable<T>`
Imposes a total ordering on the objects of each class that implements it.
	`int compareTo(T o)`
	Compares this object with the specified object for order.
		`Boolean`
		`Byte`
		`Character`
		`Double`
		`Float`
		`Integer`
		`Long`
		`Short`
		`String`

`Iterable<T>`
Implementing this interface allows an object to be the target of a for-each statement.
	`Iterator<T> iterator()`	`ArrayList`
		`HashSet`
		`LinkedList`
		`TreeSet`

`Iterator<E>`
An iterator over a collection.
The `remove` operation is optional; that is, a class can throw `UnsupportedOperationException` if `remove` is not implemented.
	`boolean hasNext()`
	`E next()`
	`void remove()`
		`Scanner`

`List<E>`
`extends Collection<E>`
*
An ordered collection.
	In addition to `Collection` methods,
	`E get(int index)`
	`int indexOf(Object o)`
	`E set(int index, E element)`
		`ArrayList`
		`LinkedList`

`Map<K,V>`
*
An object that maps keys to values.
	`boolean containsKey(Object key)`
	`boolean containsValue(Object value)`
	`V get(Object key)`
	`V put(K key, V value)`
	`int size()`
		`HashMap`
		`TreeMap`

`Runnable` The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread.	`void run()`	`Thread`

Interface	Sample methods	Implemented by

`Serializable`
Implementing this interface enables serializability of a class.
Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable.
	The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.	`ArrayList`
		`HashMap`
		`HashSet`
		`LinkedList`
		`TreeMap`
		`TreeSet`

`Set<E>`
`extends Collection<E>`
`*`
A collection that contains no duplicate elements.
	In addition to `Collection` methods,
	`boolean equals(Object o)`
	`boolean isEmpty()`
		`HashSet`
		`TreeSet`

`SortedMap<K,V>`
`extends Map<K,V>`
`*`
A map that further guarantees that it will be in ascending key order according to the natural ordering of its keys (see `Comparable`).
	In addition to `Map` methods,
	`K firstKey()`
	`K lastKey()`
		`TreeMap`

`SortedSet<E>`
`extends Set<E>`
`*`
A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the natural ordering of its elements (see `Comparable`).
	In addition to `Set` methods,
	`E first()`
	`E last()`
	`SortedSet<E> subSet(E fromEl,`
	`E toEl)`
		`TreeSet`

2.4 Collection classes

Legacy data structures such as `Hashtable` and `Stack` and `Vector` have been omitted.

Collection class	Meaning	Implements

`ArrayList<E>`	Can store a variable number of references, similar to an array.	`Collection<E>`
		`Iterable<E>`
		`List<E>`
		`Serializable`

`HashMap<K,V>`	Hash table based implementation of the `Map` interface.	`Map<K,V>` `Serializable`

`HashSet<E>`	Implements the `Set` interface, backed by a hash table.	`Collection<E>`
		`Iterable<E>`
		`Serializable`
		`Set<E>`

`LinkedList<E>`	Linked list implementation of the `List` interface.	`Collection<E>`
		`Iterable<E>`
		`List<E>`
		`Serializable`

`TreeMap<K,V>`	Tree based implementation of the `SortedMap` interface.	`Map<K,V>`
		`Serializable`
		`SortedMap<K,V>`

`TreeSet<E>`	Implements the `Set` and the `SortedSet` interfaces, backed by a `TreeMap`.	`Collection<E>`
		`Iterable<E>`
		`Serializable`
		`Set<E>`
		`SortedSet<E>`

2.5 Input and output streams and related classes

A stream is a sequence of bytes and the various stream classes provide ways of interacting with such streams. We have only listed the more commonly used input and output streams.

Classes whose names end in `Stream` handle raw data in terms of bytes. Bytes are read as `int` values in the range 0 to 255. If no byte is available because the end of the stream has been reached, –1 is returned.

Classes whose names end in `Reader` or `Writer` handle character data, represented as an `int`.

Stream methods may throw an `IOException`.

The remainder of this document is printed in a landscape format to accommodate some wide tables.

2.5.1 Input streams

We list here some of the more commonly used input streams.

`InputStream` and its descendants

Stream	**Sample methods and comments**	**Constructors**

`InputStream`
The abstract superclass of all classes representing an input stream of bytes.
Methods listed here are also available to subclasses (but may have been overridden).
	`void close()`
	`abstract int read()`
	`int read(byte[] b)`
		`InputStream()`

`FileInputStream` Obtains bytes from a file.	see `InputStream`	`FileInputStream(File f)` `FileInputStream(String name)`

`BufferedInputStream`
Adds ability to buffer input to another input stream.
	Adds methods to mark a place in a stream and return to it, while the `readLimit` is not exceeded.
	`void mark(int readlimit)`
	`void reset()`
		`BufferedInputStream(InputStream in)`

`DataInputStream`
Reads from an underlying input stream and converts to primitive types.
	Adds methods to read primitive types, for example:
	`float readFloat()`
	`boolean readBoolean()`
		`DataInputStream(InputStream in)`

`Reader` and its descendants

Classes based on `Reader` are for reading character streams.

A read character is returned as an `int` value.

Stream	Sample methods and comments	Constructors
`Reader` Abstract class for reading character streams. Methods listed here are also available to subclasses (but may have been overridden).	`abstract void close()` `int read()`	`protected Reader()`

`BufferedReader`
Read text from a character-input stream, buffering characters.
	Adds methods to mark a place in a stream and return to it, while the `readLimit` is not exceeded.
	`void mark(int readLimit)`
	`String readLine()`
	`void reset()`
		`BufferedReader(Reader in)`
		`BufferedReader(Reader in, int bufSiz)`

`InputStreamReader` A bridge from byte streams to character streams.	see `Reader`	`InputStreamReader (InputStream in)`
`FileReader` Convenience class for reading character files.	see `Reader`	`FileReader(File f)` `FileReader(String name)`

2.5.2 Output streams

Operations may throw an `IOException`. Output streams may be flushed.

`OutputStream` and its descendants

A character to be written is contained in the 16 low-order bits of a given integer value; the 16 high-order bits are ignored. (So, an `int` represents a single character to be written.)

Stream	**Sample methods and comments**	**Constructors**

`OutputStream`
The abstract superclass of all classes representing an output stream of bytes.
Methods listed here are also available to subclasses (but may have been overridden).
	`void close()`
	`void flush()`
	`void write(byte[] b)`
	`void write(int b)`
		`OutputStream()`

`FileOutputStream`
`extends OutputStream`
An output stream for writing bytes to a file.
	See `OutputStream`	`FileOutputStream(File f)`
		`FileoutputStream(String name)`
		`FileoutputStream(String name, boolean append)`

`DataOutputStream`
`extends OutputStream`
Writes primitive types to an underlying output stream.
	Adds methods for writing primitive types, for example:
	`void writeInt(int v)`
	`void writeFloat(float v)`
		`DataOutputStream (OutputStream out)`

`BufferedOutputStream`
`extends FilterOutputStream`
Adds buffering to another output stream.
	See `OutputStream`	`BufferedOutputStream(OutputStream out)`

`PrintStream`
`extends FilterOutputStream`
Adds ability to print representations of various data values.
	Adds `print` and `println` methods for primitive data types, for example:
	`print(boolean b)`
	`println(char c)`
		`PrintStream (File f)`
		`PrintStream(OutputStream out)`
		`PrintStream(OutputStream out, boolean autoflush)`
		`PrintStream(String filename)`

`Writer` and its descendants

Stream	Sample methods and comments	Constructors

`Writer`
Abstract class for writing to character streams.
Methods listed here are also available to subclasses (but may have been overridden).
	`abstract void close()`
	`abstract void flush()`
	`void write(int c)`
	`void write(String s)`
		`protected Writer()`

`BufferedWriter` Writes text to a character-output stream, buffering characters.	See `Writer`	`BufferedWriter(Writer out)`
`OutputStreamWriter` A bridge from character streams to byte streams.	See `Writer`	`OutputStreamWriter(OutputStream out)`

`PrintWriter`
Prints formatted representations of objects to a text-output stream.
	Adds `print` and `println` methods for primitives and `String`, for example:
	`print(int b)`
	`println(String s)`
		`PrintWriter (File f)`
		`PrintWriter(OutputStream out)`
		`PrintWriter(OutputStream out, boolean autoflush)`
		`PrintWriter(String filename)`
		`PrintWriter(Writer out)`

`FileWriter`
Convenience class for writing character files.
	See `Writer`	`FileWriter (File f)`
		`FileWriter(File f, boolean append)`
		`FileWriter(String filename)`

2.5.3 Standard streams

Stream	**Comments**	**Examples**
`System.in` Standard input, normally the keyboard.	`in` is a static `InputStream` in the `System` class. This stream is already open and ready to supply input data.	`Scanner scnr = new Scanner(System.in)`
`System.out` Standard output, normally the screen console.	`out` is a static `PrintStream` in the `System` class. This stream is already open and ready to accept output data.	`System.out.println("Off they go!");`
`System.err` Standard error stream, normally the screen console.	`err` is a static `PrintStream` in the `System` class. This stream is already open and ready to accept output data.	`System.err.println("bad wolf");`

2.5.4 Scanner

Stream	Sample methods and comments	Sample constructors

`Scanner`
Implements the `Iterator<String>` interface and can read text from files, input streams, strings or any object that implements the `Readable` interface.
The default delimiter for tokens is whitespace.
	`boolean hasNext()`
	`String next() //next token`
	`String nextLine()`
	Similar methods are provided for primitive types, for example:
	`int nextInt()`
	`boolean hasNextInt()`
		`Scanner(File source)`
		`Scanner(InputStream source)`
		`Scanner(String source)`

2.5.5 Sockets

Class	Sample methods	Sample constructors

`ServerSocket`
Implements a server socket which waits for requests to come in over a network.
	`Socket accept()`
	`void close()`
	`int getLocalPort()`
	`String toString()`
		`ServerSocket(int port)`

`Socket`
Implements a client socket; an endpoint for communication between two machines.
	`InputStream getInputStream()`
	`OutputStream getOutputStream()`
	`int getPort()`
	`void close()`
		`Socket(String host, int port);`

2.6 Java Swing classes

Note that most widgets have many variant constructors and methods and space would not permit listing them all. In the examples below we have simply picked the commonest constructors for each widget and some of the frequently used methods.

Top-level Swing containers – such as `JFrame`, `JDialog` and `JApplet` – are specialized components that provide a place for other Swing components to paint themselves. These classes inherit from the `Container` class, and provide methods to `add` components, with or without constraints (such as can be applied to a `BorderLayout`), as well as a method to `remove` a component.

See also the event-handling classes in Section 2.7.

Class	Sample methods	Sample constructors
`ButtonGroup` A group of radio buttons.	`Component add(JRadioButton j)` `void remove(JRadioButton j)`	`ButtonGroup ()`

`JApplet`
An applet container with support for Swing component architecture.
	`void add(Component c)`
	`void add(Component c, Object constraints)`
	`void init()`
	`void remove(Component c)`
	`void setLayout(LayoutManager m)`
	`void start()`
	`void stop()`
	`void destroy()`
		`JApplet()`
		Default layout for content pane is `BorderLayout`

`JButton` An implementation of a button that can be clicked.	`String getText()` `void setText(String text)`	`JButton ()` `JButton(String text)`

Class	Sample methods	Sample constructors

`JCheckBox`
An implementation of a check box that can be selected or deselected.
	`Object[] getSelectedObjects()`
	`String getText()`
	`boolean isSelected()`
	`void setSelected(boolean b)`
		`JCheckBox (String text)`

`JComboBox`
A component that combines a button or editable field and a drop-down list.
	`void addItem(Object o)`
	`Object getItemAt(int index)`
	`Object getSelectedItem()`
	`Object[] getSelectedObjects()`
	`void setSelectedItem(Object o)`
		`JComboBox()`
		`JComboBox(Object[] items)`

`JFrame`
A top-level container; the window used in a graphical user interface.
	`void add(Component c)`
	`void add(Component c, Object constraints)`
	`Container getContentPane()`
	`void paint(Graphics g)`
	`void remove(Component c)`
	`void repaint()`
	`void setDefaultCloseOperation(int operation)`
	`void setJMenuBar(JMenuBar m)`
	`void setLayout(LayoutManager m)`
	`void setTitle(String title)`
	`void setVisible(boolean value)`
	`void update(Graphics g)`
		`JFrame (String title)`
		Default layout is `BorderLayout`

`JLabel` A display area for a short text string.	`String getText()` `void setText(String text)`	`JLabel (String text)`

Class	Sample methods	Sample constructors
`JList` A component that allows the user to select one or more objects from a list. Occupies a fixed number of lines.	`Object[] getSelectedValues()` `int[] getSelectedIndices()`	`JList (Object[] listData);`

`JMenu`
An implementation of a pull-down menu that can be held in a `JMenuBar`.
	`JMenuItem add(JMenuItem j)`
	`JMenuItem add(String s)`
	`void insert (String s, int pos)`
	`void remove (int pos)`
		`JMenu (String text)`

`JMenuBar` An implementation of a menu bar acting as a holder for menus.	`JMenu add(JMenu c)`	`JMenuBar ()`

`JMenuItem`
An implementation of an item in a menu.
	`void add(Component c)`
	`void init(String text, Icon icon)`
	`void setEnabled(boolean b)`
		`JMenuItem (String text)`

`JPanel`
A container used to place widgets and which can be added to a `JFrame` or to another panel.
	`Component add(Component c)`
	`void add(Component c, Object constraints)`
	`void paintComponent(Graphics g)`
	`void remove(Component c)`
	`void repaint()`
	`void setLayout(LayoutManager m)`
		`JPanel ()`
		`JPanel (LayoutManager layout)`
		Default layout is `FlowLayout`

Class	Sample methods	Sample constructors

`JRadioButton`
An implementation of a radio button, used in conjunction with a ButtonGroup so that only one radio button at once can be selected.
	`void setText(String text)`
	`boolean isSelected()`
	`void setSelected(boolean b)`
		`JRadioButton (String text)`
		`JRadioButton (String text, boolean selected)`

`JScrollBar`
An implementation of a scroll bar with a slider that can be moved.
	`int getMinimum()`
	`int getMaximum()`
	`int getValue()`
	`int setMinimum()`
		`JScrollBar ();`
		`JScrollBar (int orientation);`

`JScrollPane` A scrolling pane that includes horizontal and vertical scroll bars and can contain a list or text area.	Used to add scrolling ability to a `Component` such as a `JList` or `JTextArea`.	`JScrollPane (Component c)`

`JTextArea`
An implementation of a multi-line holder of text.
	`void append(String s)`
	`int getLineCount()`
	`void setColumns(int c)`
	`void setRows(int r)`
	`void setText(String s)`
		`JTextArea(String s)`
		`JTextArea (String s, int rows, int cols)`
		`JTextArea (int rows, int cols)`

`JTextField`
An implementation of a single line holder of text.
	`String getText()`
	`void setText(String s)`
		`JTextField()`
		`JTextField(int columns)`
		`JTextField(String text)`

2.7 Event handling

Relevant components have an `add` method formed from the name of the listener, for example, `addActionListener` or `addAdjustmentListener`. The `ActionEvent` class provides a method `Object getSource()` method to return a reference to the object that produced an event.

Interface	Methods	Events generated by the following
`ActionListener`	`void actionPerformed(ActionEvent e)`	Buttons, lists, menu items and text fields.
`AdjustmentListener`	`void adjustmentValueChanged(AdjustmentEvent e)`	Scroll bars.

`ComponentListener`	`void componentHidden(ComponentEvent e)`
	`void componentMoved(ComponentEvent e)`
	`void componentResized(ComponentEvent e)`
	`void componentShown(ComponentEvent e)`
		Visual components; for example, being resized or hidden.

`ContainerListener`	`void componentAdded(ContainerEvent e)` `void componentRemoved(ContainerEvent e)`	Containers such as frames; for example, when a component is added or removed.
`FocusListener`	`void focusGained(FocusEvent e)` `void focusLost(FocusEvent e)`	Components coming into focus or going out of focus.
`ItemListener`	`void itemStateChanged(ItemEvent e)`	Check boxes, choices and lists.

`KeyListener`	`void keyPressed(KeyEvent e)`
	`void keyReleased(KeyEvent e)`
	`void keyTyped(KeyEvent e)`
		Keys being pressed or released.

`MouseListener`	`void mouseClicked(MouseEvent e)`
	`void mouseEntered(MouseEvent e)`
	`void mouseExited(MouseEvent e)`
	`void mousePressed(MouseEvent e)`
	`void mouseReleased(MouseEvent e)`
		Actions such as clicking or moving a mouse.

`TextListener`	`void textValueChanged(TextEvent e)`	Text components such as text fields and text areas.
`WindowListener`	`void windowClosing(WindowEvent e)`	Windows being opened or closed.

2.8 Layout managers

Layout manager	Effect	Sample code

absolute positioning	Components are placed manually. The layout manager is specified to be `null`.
	`setBounds` is used to place and size the component.
		`holder.setLayout(null);`
		`JButton jb = new JButton();`
		`holder.add(jb);`
		`jb.setBounds(10,10,20,20);`

`BorderLayout`	Components are placed at north, south, east, west, or centre.	`BorderLayout()`
		data includes static constants
		`BorderLayout.NORTH`
		`BorderLayout.SOUTH`
		`BorderLayout.WEST`
		`BorderLayout.EAST`

`FlowLayout`	Components are arranged like words in a paragraph, flowing to the next line if they will not fit on the current one.	`FlowLayout()`
`GridLayout`	Components are arranged in a grid of rows and columns (arguments are in that order).	`GridLayout(int rows, int cols)`

2.9 Graphics

Sample methods	Meaning
`drawRect(int x, int y, int width, int height)`	Draws the outline of the specified rectangle. The left and right edges of the rectangle are at `x` and `x + width`. The top and bottom edges are at `y` and `y + height`. The rectangle is drawn using the graphics context's current colour.
`drawLine(int x1, int y1, int x2, int y2)`	Draws a line, using the current colour, between the points `(x1, y1)` and `(x2, y2)` in this graphics context's coordinate system.
`drawOval(int x, int y, int width, int height)`	Draws the outline of an oval. The result is a circle or ellipse that fits within the rectangle specified by the `x`, `y`, `width`, and `height` arguments.
`drawString(String s, int x, int y)`	Renders the text of the specified `String`, using the current text attribute in the graphics context, starting from (`x`, `y`)
`fillOval(int x, int y, int width, int height)`	Fills an oval bounded by the specified rectangle with the current colour.
`void setColor(Color c)`	Sets this graphics context's current colour to the specified colour. All subsequent graphics operations using this graphics context use this specified colour. Colours are static constants, for example `Color.GREEN`, `Color.RED` etc.

2.10 Some common methods of the `MIDlet` class

Method	Meaning
`protected abstract void startApp()`	Signals the `MIDlet` that it has entered the *Active* state from the *Paused* state.
`protected abstract void pauseApp()`	Signals the `MIDlet` to enter the *Paused* state from the *Active* state.
`protected abstract void destroyApp(Boolean b)`	Signals the `MIDlet` to terminate, releasing its resources, and enter the *Destroyed* state.

Printed in the United Kingdom.