The Open University

Mathematics and Computing: Level 2
M255 Object-oriented programming with Java

M255
Glossary

WEB 86853 0
1.1

A

abstract class A class that defines a **common message protocol** and common set of **instance variables** for its **subclasses**. In Java an abstract class cannot be **instantiated**.

abstract method A **method** declared as `abstract`. Abstract methods have no bodies. They must be implemented in any **concrete subclasses** of the **class** in which they are specified.

abstraction A description that focuses on the essential features of a problem and ignores other details.

acceptance testing Testing that software fulfils a customer's requirements.

access modifier One of three Java keywords (`public`, `private`, `protected`) which specify the **visibility** of **variables** and **methods**.

accessor message An accessor message is either a **getter** or a **setter** message. For example, the messages `getPosition()` and `setPosition()` are accessor messages for the **instance variable** `position` held by instances of the `Frog` class. The getter message `getPosition()` returns the value of the instance variable `position`, while the setter message `setPosition()` changes the value of `position`.

accessor method A method that implements an **accessor message**. See **getter method** and **setter method**.

activation rectangle An element in a **sequence diagram** that represents a period during which a particular **object** is active.

actual argument The value a **formal argument** holds at **run-time**. For example, when the **message** `setPosition()` is sent to a `Frog` object an actual argument is used. This actual argument must be type compatible with the formal argument in the **method header** of the corresponding **method**. So in the **message-send**

```
frog4.setPosition(3);
```

the integer `3` is the actual argument for the message. On receiving the message `setPosition(3)`, the `Frog` object referenced by `frog4` causes the method `setPosition()` to be executed with its formal argument `aPosition` set to the value of the message's actual argument, in this case `3`.

actual results Given a set of **test data**, the result of executing a **method** or **program**.

adaptive method A method of **software development** which embraces change by building space for it into the schedule.

alpha testing Testing which is done before a product is available publicly.

analysis Analysis involves analysing the software **requirements** to develop a detailed understanding, in computing terms, of what the software has to do. The outcome is a **requirements specification** document.

application Software that turns your computer into a specialised computer, such as a word processor or web browser.

application domain See **problem domain**.

argument Some **messages** require information. For example, when requesting a `Frog` object to change its colour to that of another `Frog` object, it is necessary to specify the other `Frog` object. This is seen in the **message-send** `frog1.sameColourAs(frog2)`. A message can have zero, one or more arguments. The arguments (if any) used in a message are known as *actual arguments*, since they contain actual values or references to actual objects that will be passed to the corresponding method via its **formal arguments**. The actual arguments must match the formal arguments in number and type, and must appear in the same order.

array An **indexable, fixed-size collection** whose **elements** are all of the same type.

assertion A statement which tests whether an expected value matches an actual one.

assign See **assignment**.

assignment When using objects, assignment is the process which results in the **variable** on the left-hand side of the assignment operator referencing the object returned by the expression on the right-hand side (this is called assignment using **reference semantics**). When using values of **primitive data types**, assignment is the process that results in the variable on the left-hand side containing a copy of the *value* returned by the right-hand side (this is referred to as assignment using **value semantics**).

assignment statement A **statement** that tells Java to make a **variable** reference a particular **object** or to hold a particular **primitive** value (see **assignment**).

attribute Some property or characteristic of an **object**, such as `position` for `Frog` objects, or `balance` for `Account` objects.

attribute value The current value of an **attribute**. For example, a `Frog` object has the attributes `colour` and `position`. The attribute `colour` of a particular object might have the value `OUColour.BLUE` and the attribute `position` might have the value `1`.

autoboxing **Collections** that are part of the **Collections Framework** can only contain **objects** – not **primitive types**. Before Java 1.5, this meant that the programmer had to manually wrap each primitive value in an instance of its corresponding **wrapper class** before it could be added to a collection. Conversely, on removal from a collection, each primitive value had to be removed from its wrapper before use. This process was laborious, tedious and error-prone, and is still widespread in older code. However, since the advent of Java 1.5, primitive values may now in effect be added to and accessed from collection classes, since the **compiler** now ensures that the above process takes place automatically. This automatic process of wrapping and unwrapping primitive values when needed is called autoboxing.

B

behaviour This term is used to describe the way an **object** responds to the **messages** in its **protocol**.

beta testing Testing of a trial version of a product by potential users, who provide feedback to the developers.

binary digit Either of the two digits 0 and 1 in the binary number system. Binary digits are used for the internal representation of numbers, characters and instructions. The binary digit is the smallest unit of storage.

bit See **binary digit**.

black box testing Testing something against specification, without any knowledge of its inner workings.

block See **statement block**.

Boolean condition A **Boolean expression** used to control the conditional execution of a **statement block**.

Boolean expression An expression that evaluates to either `true` or `false`. Boolean expressions can be simple or complex and can involve a number of **variables**.

Boolean operators Operators used to combine simple **Boolean expressions** to form more complex Boolean expressions, which in turn can be combined with other Boolean expressions. They are also known as logical operators.

boundary error A programming error which causes an incorrect path to be taken through code.

buffer (reading/writing files) An area used for temporary storage as data is transferred between a data source and a data sink.

buffer `(StringBuilder)` A sequence of unfilled **components** that allows a `StringBuilder` object to grow.

bug The cause of a **run-time error**.

bulk operation **Iteration** offers the programmer the opportunity to access or process each **element** of a **collection** in turn and deal with each as they think fit. By contrast, bulk operations are operations that act on an entire collection without requiring the programmer to focus on individual elements.

bytecode Bytecode is the intermediate code produced by the Java **compiler**. In BlueJ, compilation is done when the Compile button is pressed. This will create a bytecode file, for example `Frog.class`, from the source code file `Frog.java`. The bytecode file is portable, because each computer that can run Java programs has a Java Virtual Machine – a program itself – that understands bytecode and converts it into the machine code required for that particular computer.

C

CASE (computer-aided software engineering) tool A software tool used to help in some aspect of **software development**.

cast A way of modifying the type of a **variable** or **expression** 'on the fly'.

catch The process of catching an **exception** – see `try–catch` statement.

chained See **constructor chaining**.

checked exception An **exception** that the **compiler** requires the **programmer** to handle (with a `try–catch` statement) if they write code that executes a method that can throw such an **exception**.

class A class is a template that serves to describe all **instances** (**objects**) of that class. It defines what **attributes** the objects should have and their **protocol** – what **messages** they can respond to.

Instances of the same class have the same attributes, which are **initialised** in the same way. They have the same **instance** protocol and respond in the same way to each message.

class header The line in a class definition which provides crucial information such as its name, **access modifier**, name of a class from which it extends and name(s) of any **interface(s)** it **implements**. Example usage:

```
public class WeatherFrog extends Frog implements WeatherClient
```

class method A class method is a **method** declared with the keyword `static`. Class methods are associated with a **class** rather than with any **instances** of a class. In general, a class method has no information about the existence of, or state of, any instances of its class. Class methods are invoked directly on the name of the class, and not by sending a **message**. In Java, classes (as opposed to their instances) are not **objects**, so class methods are *not* object-oriented and do not exhibit **inheritance** or **polymorphism**.

class variable A class variable is a **variable** declared with the keyword `static`. A class variable is associated with a **class** rather than with any of its **instances**. A class only ever has one copy of each of its class variables. In Java, classes (as opposed to their instances) are not objects hence class variables do not take part in **inheritance**. See **qualified** for details of how class variables are accessed.

client This term has two main meanings in the context of **software development**: (i) the object in a **collaboration** which requests a service; (ii) the person(s) commissioning software.

code-based testing tool A **testing tool** that automatically analyses code and produces test cases.

coding tool A **CASE tool** that aids writing or running code.

collaboration One **object** requesting a service from another object.

collaborator A participant in a **collaboration**.

collection A collection consists of a number of, possibly zero, **objects** or **primitives**. The objects or primitives within a collection are referred to as **elements**.

collection classes/**Collection Classes** When capitalised, this phrase usually refers to collection classes that are part of the **Collections Framework** – by this fact *excluding* arrays. When used in lower case, the same phrase generally means any class that implements a collection (in the looser English language sense) – which *includes* arrays.

Collections Framework Shorthand for the **Java Collections Framework**.

comment A comment is a piece of text in **program** code that is ignored when executing the code. In Java multi-line comments are delimited by `/*` and `*/`. Single line comments are simply preceded by `//`. A comment can generally be placed anywhere in the code of a class, with the exception of **method** comments – method comments are placed between `/**` and `*/` and must appear *immediately before* the method header.

common message protocol A set of **messages** shared by a number of classes. Often used to describe the set of messages provided by an **abstract class** for its **subclasses**.

`Comparable` An **interface** that defines the header for a single **method**: `compareTo()`. **Classes** which **implement** this interface enable their **instances** to have a **natural order** and can therefore be added as **elements** to sorted **collections**.

`compareTo()` The sole method of the `Comparable` interface. Allows **classes** of **objects** implementing this **interface** to be sorted (i.e. gives them a **natural ordering**).

compiler A **program** that translates **source code** into **bytecode** or machine code.

compile-time error Errors that are to do with the form of the text as determined by the rules for a given programming language. These are detected and reported during the early stages of compilation (see **compiler**). The compilation does not proceed to code generation if an error is detected.

component (array) The memory location at which an **element** (or a reference to it) of an **array** is stored.

component (software) See **software component**.

component type (arrays) The declared type of the **components** of an **array**. This is stated on the declaration and creation of the **array**.

compound expression An **expression** built up using other sub-expressions; for example, the following is a compound expression: `(3 + 2) * (6 - 3)`.

computation error A programming error which causes data to be processed incorrectly; for example, the wrong calculation may be performed on the data.

concatenation The joining of two strings. In Java the string concatenation operator is `+` (the plus sign). For example, `"Milton " + "Keynes"` evaluates to `"Milton Keynes"`.

concrete class A class which is not **abstract**; a class for which instances can be created.

condition See **Boolean condition**.

conditional selection The use of `if` statements to select and execute alternative **statement blocks** based upon the value of a **Boolean condition**.

constant A **constant** is a **variable** whose value is fixed and unchangeable. Normally the keyword `final` is used to make a value into a constant. In addition, where only a single value is needed for a class, constants are typically declared as `static`. However, `static` is not always used for constants, as sometimes a situation needs to be modelled where each instance of a class has its own *different* constant value, such as a serial number.

constant instance variable Constants are usually declared as `final static` variables. However, sometimes it makes more sense to define a constant as a `final` **instance variable**. See **constant** for more information.

constructor A special type of **message** used to **initialise** a newly created **object**.

constructor chaining The process whereby **constructors** use `super()` to invoke each other up the inheritance hierarchy.

D

data field A synonym for **instance** variable (also a synonym for class variable).

data hiding This is where an **object** is treated as a black box, with access to the **encapsulated** data (the **instance variables**) being possible *only* through a limited set of **methods**, i.e. only an object's own methods are allowed to access the value of an instance variable (either to change it or return it).

data member See **data field**.

debugging Identifying the cause of faults (bugs) in software and correcting them.

delta A tolerance used for comparing floating-point numbers, which are not represented exactly in the computer, so that two values which should be equal may differ marginally. We set a small delta and then consider that two floating-point numbers are the same if they differ by less than the delta.

design Design involves deciding how the **software** will meet the specified **requirements**.

design principle In contrast to a guideline or suggestion, the word *principle* is reserved for recommendations that apply universally or nearly universally. In this course, the term **design principle** is applied to principles governing how code should be organised.

design tool A **CASE tool** that aids some aspect of **design**.

designer A developer whose role is to work on the **design** stages of a project.

designing dynamic models Determining what interactions among objects will achieve the tasks required of the **software**.

destructive operation An operation on a **collection** (i.e. a **message-send**) that changes the contents in some way is said to be destructive**.** It is not always obvious whether a message is destructive or **non-destructive** – for example, a **bulk operation** that might be expected to change the contents may leave it unaffected but return an altered copy, or vice versa.

detailed design and implementation Deciding what existing **classes** can be reused and what programming constructs are appropriate as well as writing the actual code.

developing a structural model Analysing the **requirements** to determine the **classes** and connections between them that are appropriate for the area the software is being written for, thus defining a structure for the **software**.

developing a user interface Designing the user interface and determining how it will communicate with the **domain model**.

dialogue box A mechanism whereby users can be given information by the system or provide information to the system on request. Dialogue boxes are implemented by **class methods** of `OUDialog`. The signatures of these methods are:

```
alert(String)
confirm(String)
request(String)
request(String, String)
```

difference Difference is a mathematical operation on two sets, that in Java is implemented by a `removeAll()` message. Given two sets, `set1` and `set2`, the **message-send** `set1.removeAll(set2)` would remove from `set1` all the **elements** contained by `set2`. The term 'difference' is used to refer to both the result and the operation.

direct interaction **Direct interaction** is not a formal technical term, but a descriptive phrase used to describe a situation where one **object** has a reference to another, and this reference is used to affect the state or behaviour of the other object. Contrast with **indirect interaction**.

direct subclass A **class** is a direct subclass of another class if it is directly below that class in the class hierarchy. In Java a class extends the class of which it is a direct **subclass**.

direct superclass A **class** is a direct superclass of another class if it is directly above it in the class hierarchy.

domain See **problem domain**.

domain model That part of the software that models the **problem domain** and is not directly concerned with how communication with the user is achieved.

dynamic compilation A compilation technique (used by the Java environment) generating real machine code from **bytecode** (intermediate code). A chunk of bytecode is compiled into machine code just prior to being executed. (This is different from and faster than the piecemeal operation of an interpreter.) The real machine code is retained so that subsequent execution of that chunk of bytecode does not require the translation to be repeated.

dynamic model An illustration of events occurring in executing **software** over time.

dynamic semantic error A **semantic error** that cannot be detected by the **compiler**, but which can be detected by the JVM or a method at **run-time** and which results in an **exception** being thrown.

E

effective length The number of meaningful **elements** that a **fixed-size collection** actual holds.

element The name given to a primitive value or a reference to an **object** stored in an **array component**.

encapsulation **Objects** allow you to encapsulate data by incorporating into a single entity (the object) both the data (**instance variables**) and the **behaviour** (**methods**) defined for that data. The concept of encapsulation is very powerful because it allows an efficient division of labour in large software projects. Each team member can work in isolation on one or more classes. The only things that team members need to know about other classes are the names and specifications of the methods.

entry An entry in a **map** means a **key**–**value pair** – not just the **key** or the value.

`equals()` A method inherited by all classes from `Object`. In the `Object` class the **method** simply compares the **receiver** with the **argument** using the == operator and so will return true if they both reference exactly the same **object** – in other words, they both point to the same address in memory. However, the method is frequently **overridden** to define versions of equality appropriate to particular classes and particular purposes. For many classes, such as `String`, the method `equals()` is defined effectively to mean 'has the same state as'. More generally, `equals()` is coded to mean 'should be regarded as equal for our purposes'.

equivalence class All the possible input values to a block of code (for example, a **method**) can be divided into mutually exclusive sets of values. Each set of values is called an equivalence class. Each value in an equivalence class can be expected to be processed by that block of code in the same manner. The corollary is that values from different equivalence classes will be treated in a different manner (i.e. be processed by a different branch of the code).

exception An **object** that is thrown by a **method** or the JVM as the result of a **run-time error**. The exception object holds details of what went wrong allowing the exception handler which catches the exception to take appropriate action.

exception handling The programmed catching of an **exception** and the subsequent execution of reliable code to abandon execution, for example, or to restore the **software** to a meaningful state.

expected results Given a set of **test data**, results that a **method** or **program** should produce according to its specification.

expression Code that evaluates to a single value. Expressions are formed from **variables**, operators and **messages**.

F

field See **data field**.

`File` An **instance** of this **class** contains the pathname to a file or folder in a system-independent format. The file specified need not exist; instead the pathname may point to a potential new file or folder.

`final` A keyword that indicates that an **instance variable** may not be changed once it has been assigned an initial value.

fixed-size collection A **collection** whose number of **elements** is fixed on creation. Such a collection can neither grow nor shrink.

`foreach` statement A **statement** that enables **iteration** through every **element** of a **collection**.

formal argument A typed **identifier**, declared in a method's **argument** list (between parentheses), that at **run-time**, when the **method** executes, will hold a value (or reference an **object**) that is passed into that method by the **receiver** of a corresponding **message**. For example, in the following **method header**:

```
public void setPosition(int aPosition)
```

the formal argument is the identifier `aPosition` and it is declared as being of type `int`. See **actual argument**.

formatting guidelines A set of guidelines which specify how program code should be laid out.

G

garbage collection The process of destroying **objects**, which have become unreachable because they are no longer referenced by variables, in order to reclaim their space in memory. In certain programming languages, including Java, this process is automatic.

getter message A **message** that returns as its **message answer** the value of one of a **receiver's attributes**. See **setter message** and **accessor message**.

getter method An **accessor method** whose purpose is to return the value of an **instance variable** as its **message answer**.

H

hash bucket A hash bucket is a 'compartment' used in the internal implementation of a **set**. The set uses the **hash code** of each incoming element to decide which hash bucket to store it in.

hash code In Java, a hash code is an integer that can be calculated for any **object**, by sending it the **message** `hashCode()`. This **method** is inherited from `Object`, but in general must be **overridden** if `equals()` has been overridden. Hash codes are used by **sets** and related **collections** to allow them to store **elements** efficiently and to check rapidly for duplicates. If a **class** redefines `equals()`, then the `hashCode()` method for that class must take the definition into account; otherwise the type will not behave properly in collections.

helper method A **method** which carries out some subsidiary task, such as a calculation, for another method. Helper methods would normally be private.

high-level language A language (for example, Java) whose structure reflects the requirements of the problem, rather than the facilities actually provided by the hardware. It enables a **software** solution to a problem, or a simulation of an aspect of reality, to be expressed in a hardware-independent manner.

homogeneous collection A **collection** in which all the **elements** are of the same type.

I

identifier (modelling) A label chosen to identify an **object** in the **software**.

identifier (programming) The name of a **variable**.

identity Two **variables** have the same identity if they both reference the same **object** or **primitive**.

immutable An **object** or **primitive** that cannot be changed.

implementation Translating a **design** into program code in some suitable programming language.

`implements` A keyword in Java used in a class header to specify that the class implements a particular **interface**. Example usage:

```
public class SomeClass implements SomeInterface
```

index An integer that is used to access the **elements** of an **indexable collection**.

indexable collection A **collection** in which every **element** is accessed by an integer **index**.

indirect interaction **Indirect interaction** is not a formal technical term, but a descriptive phrase used to describe a situation where one **object** affects the state or **behaviour** of the other object, but without actually having a reference to that object – i.e. some intermediary object is used. Contrast with **direct interaction**.

indirect subclass A **class** is an indirect subclass of another class if it **inherits** from that class via one or more intermediate classes.

indirect superclass A **class** is an indirect superclass of another class, if it is above it in the class hierarchy but not directly above it.

inheritance A relationship between **classes** by which they are organised into a hierarchy. Classes lower in the hierarchy are said to inherit all the **methods** and **variables** from classes higher in the hierarchy (though they may also define additional methods and variables).

initialisation The setting of **instance variables** of **objects** to appropriate values following their creation.

`InputStream` The base **class** of a hierarchy of classes including `FileInputStream`, `BufferedInputStream` and `ObjectInputStream`, which are used to read 8-bit byte streams. `InputStream` classes are used to read binary data.

inspector An inspector is a tool used in M255 to look at the value held (or **object** referenced) by a **variable** declared in the OUWorkspace. In the case of a variable referencing an object an inspector will show the internal state of that object, listing its **attributes** and their current values.

instance An **object** that belongs to a given **class** is described as an instance of that class.

instance method The code that is executed as the result of a **message** being sent to an object.

instance variable A **variable** whose identifier and type is common to all the instances of a **class** but whose value is specific to each instance. Each instance variable either contains a reference to an **object** or contains a value of some **primitive** type. For example, `Frog` objects have the instance variables `colour` and `position`. The values of the instance variables of a particular object represent the state of that object.

instantiate Create an **instance** (of a **class**). In Java this must be a **concrete class**. **Abstract classes** and **interfaces** cannot be instantiated.

integrated development environment (IDE) A software tool that supports the construction, compilation and execution of a program. BlueJ is an example of an IDE that supports the development of programs in Java and includes libraries of **classes** and facilities for **debugging** and **program design**.

integration testing Testing that a group of **components**, already tested in isolation from one another, work together in the way intended.

interface (Java specific) An interface specifies a list of **methods** that a group of unrelated **classes** should implement, so that their instances can interact together by responding to a common subset of **messages**. An interface only lists **method headers** (no method code), cannot declare any **instance variables** and cannot be **instantiated**. Classes implementing an interface must provide implementations for all the methods specified by that interface.

intermediate code See **bytecode**.

intersection Intersection is a mathematical operation on two or more **sets** that results in a set containing only all common elements. In Java, intersection is implemented by a `retainAll()` message. Given two sets, `set1` and `set2`, the **message-send** `set1.retainAll(set2)` would remove from `set1` any **elements** that were not also contained by `set2`. The term intersection is used to refer both to the result and to the operation.

iteration (programming) Also referred to as repetition. The repeated execution of a **statement block** for as long as some **condition** continues to be true.

iteration (software development) One cycle through the phases involved in an **iterative method**.

iterative method An adaptive method of **software development** in which phases are repeated iteratively in a systematic manner.

J

.jar file A file that 'zips up' the individual files that contain the different Java classes that make up a Java program.

Java Collections Framework The Collection Framework includes a set of **collection interfaces** (`Collection`, `Set`, `List`, `Map` and others), a set of collection **classes** (e.g. `HashSet` and `HashMap`) that implement them, and some supporting utility classes (e.g. `Collections`). The framework also includes a set of associated recommendations about constructors and about expected behaviours in common. Arrays are *not* part of the Collections Framework.

Javadoc A program that comes with Java. The Javadoc program picks up information from specially formatted **comments** and other parts of the class code such as the **constructor** and the **method headers**. These are all used to create an HTML file, which describes the class in a standard way. This description is aimed not at the Java compiler, but at human readers (and possibly the writer of the code at a later date, when he or she might well have forgotten what the methods do).

`java.util.Arrays` A Java utility class that contains static **methods** for manipulating **arrays**.

just-in-time compilation See **dynamic compilation**.

K

key Some **collections** of information are set up for quick convenient access using some pre-chosen part of the information stored – e.g. looking up telephone numbers using names. In such a situation, a key is something that can be used to uniquely identify any **object** from a given collection. What constitutes a good key may depend on the purpose of the collection.

key–value pair Where a **collection** (such as a map) is organised for looking up values using a single kind of **key**, the collection can be viewed as a set of key–value pairs. Each entry in a `Map` is a key–value pair. For example, a key might be a person's name, and its associated value their telephone number.

L

length The number of **elements** that an **array** can hold.

lifeline An element in a **sequence diagram** that represents the time during which an object exists.

link A connection between two **objects**.

list A list is an **ordered collection** of variable size that permits duplicate elements.

literal A comprehensible textual representation of a **primitive** value or **object**. For example, `'X'` is a `char` literal, `4.237` is a `double` literal and `"hello there!"` is a `String` literal.

local variable A **variable** declared inside a **method** body, and whose **scope** is restricted to that method.

logical error The result of code not correctly implementing the specification of particular problem. Although the code is both **syntactically** and **semantically** correct, it does not behave as expect at **run-time**.

low-level language A language written for direct programming of a computer's hardware. Each type of computer hardware needs its own low-level language.

M

`main()` A public static method that all Java programs must include in order to be executed independently of an environment such as BlueJ.

maintenance The phase of a **software development** process associated with keeping the software working to the satisfaction of its users.

map An unordered collection of **key**–**value** pairs. Duplicate keys are not allowed as they are used to look up their associated values. A **map** has a dynamic size (i.e. it can grow and shrink as key–value pairs are added and removed).

member The term **member** is a convenient word sometimes used to cover all of the four following categories: **class variables**, **class methods**, **instance variables** and **instance methods**.

message A message is a request for an **object** to do something. The only way to make an object do something is to send it a message.

For example, the position of a `Frog` object changes when it is sent the message `left()` or `right()`; to obtain information on the value of a `Frog` object's `colour` **attribute**, you send it the message `getColour()`.

message answer When a **message** is sent to an **object** then, depending on what the message is, a message answer may be returned. A message answer is a value or an object; it is not a message.

Sometimes a message answer is used, sometimes it is ignored. A message answer may be used subsequently as the **receiver** or **argument** of another message.

Enquiry messages (**getter messages**) often return the value of an **attribute**, as with the message `getColour()`, which returns a value such as `OUColour.GREEN`.

message expression A **message-send** which evaluates to a value, i.e. the message returns an answer.

message name The name of a **message** does not include any arguments. For example, the name of the message `left()` is `left()`, and the name of the message `upBy(6)` is `upBy()`.

message-send The code that sends a **message** to an **object** – for example, `frog1.right()`, which consists of the **receiver** followed by a full stop and then the **message**.

method The code that is invoked by the Java Virtual Machine at run-time when an **object** receives a **message**.

method body That part of a **method** enclosed by braces that follows the **method header**.

method header A method header consists of an optional **access modifier** (e.g. `public`), a return value (e.g. `int` or `void`) and a name (e.g. `setPosition`) followed by any **formal argument** declarations enclosed in parentheses (e.g. `(int aNumber)`). For example, the method header for a method whose name is `setPosition()` is `public void setPosition(int aNumber)`.

method invocation At **run-time**, selecting and executing a **method** when an **object** receives a **message**.

method signature The name of the **method** together with the parentheses and the types of any **arguments**. For example, the signature for the `setPosition()` method in the `Frog` class is `setPosition(int)`.

microworld A computer-based simulation with opportunities for manipulation of content and practice of skills.

model (noun) See **domain model**.

model (verb) To simulate an entity in the **problem domain**.

modelling language A specification of how **models** should be constructed so that their meaning is unambiguous.

N

natural ordering In Java, an **element** type is said to have a natural ordering if it implements the **interface** `Comparable` and consequently has a `compareTo()` method. For numbers and strings, the natural ordering corresponds simply to everyday numerical order and alphabetical order.

network computing The theory of connecting computers together, and the use of such a 'network'. A network may be local (for example, in an office or home) or global (for example, the Internet).

`new` An operator used to create an **object** – used in conjunction with a **constructor**.

non-destructive operation An operation on a **collection** that does not change the contents in any way is said to be non-destructive.

`null` A special value in Java which indicates that a variable of an **object** type does not currently hold a reference to an object.

O

`Object` Top-level class in Java. Every class in Java is either a **direct** or an **indirect subclass** of `Object`.

object An **object** is a **software component** that has a unique **identity** and responds to **messages**. Each object has **state** and responds to a particular set of **messages** (its **protocol**). Thus a `Frog` object (which has little resemblance to a real-world frog) holds information on its position and colour as values of its **attributes** `position` and `colour`.

object diagram A diagram of **objects** and the **links** between them.

object technology A synonym for **object-oriented technology**.

object-oriented technology The technology associated with viewing **software** as being made up of **objects**.

object-state diagram An object-state diagram represents an **object**. It shows the **class** of the object, its **state** in terms of attribute values, and its **protocol**.

Observer pattern A **software pattern** in which one **object** (the observer) is automatically kept informed whenever another object (the subject) changes **state**.

OMG (**Object Management Group**) A consortium of computing companies which sets standards across the software industry, including the **UML standard**.

operating system The **software** that manages the resources of a computer, including controlling the input and output, allocating system resources, managing storage space, maintaining security, and detecting equipment failure.

orchestrating instance An **orchestrating instance** is a separate **object** used to tie together the different parts of a complicated interaction that do not seem to belong to any single one of the objects involved.

ordered collection An ordered collection is a type of **collection** where each **element** in the collection has a well-defined place, as indicated by its index number. Thus, an ordered collection allows us to access and find the first, second or last element etc. However, the order does not have to reflect anything to do with the elements themselves. It is simply determined by where each element has been placed in the list. This contrasts with a **sorted collection**.

`OutputStream` The base **class** of a hierarchy of classes including `FileOutputStream`, `BufferedOutputStream` and `ObjectOutputStream`, which are used to write 8-bit byte streams. `OutputStream` classes are used to write **binary data**.

overload-resolution When a **method** is **overloaded**, the **compiler** must decide which **method signature** the JVM should use to select a method at **run-time**. The process of picking the best match from a set of candidate methods is called overload-resolution.

overloading A **method** is said to be overloaded when there are other methods, defined in the same **class**, or **inherited** from some **superclass**, with the same name but a different **method signature**, i.e. different types and/or numbers or **arguments**.

overriding The process of redefining (replacing) a **method inherited** from a **superclass** so as to cause it to have different behaviour. A method which overrides a method from a superclass has the same **signature** as the superclass method.

P

parameter A synonym for **argument**.

parse See **parsing**.

parser That part of the **compiler** which checks **source code** for **syntactic** and **semantic** correctness.

parsing The process of deciding whether the input text is a 'sentence' of a given language and obeys its **syntax** and **semantic** rules.

pattern In programming, an established and well-tried approach, technique, algorithm or coding idiom that can be used as a **model**. (In object-oriented programming, pattern is also used to refer to particular named design structures involving a number of classes and modelling a solution to a common software design problem.)

peripheral device Any part of the computer that is not part of the essential computer (i.e. the CPU and main memory). The most common peripherals are input and output (I/O) devices such as the mouse and keyboard, and storage devices such as hard disks and CD/DVD drives.

persistence The ability of **objects** or other data to continue in existence after a **program** has stopped executing.

phase A stage of **software development**.

polymorphic method A **method** with the same **signature** as some other method, but which defines different **behaviour**. For example the method `home()` defines different behaviour for `Frog`, `Toad` and `HoverFrog` objects.

polymorphic variable A **variable** which can reference **objects** of different types. In object-oriented languages, all variables declared of some object type or **interface** type are potentially polymorphic as they can be used to reference objects which are **subclasses** of that declared type or objects whose **class** implements that interface. See **substitution**.

polymorphism Any **message** to which **objects** of more than one **class** can respond is said to be polymorphic or to show polymorphism.

For example, both `Toad` and `Frog` objects respond to the message `left()`, but with different **behaviours**. They also respond to the message `green()`, with identical behaviours.

predictive method A **software development** method that is largely based on the **waterfall method** and therefore benefits from simplicity of planning, and predictability.

primary versus secondary sort Sometimes a **collection** should be sorted primarily according to a particular **key**, but in the case of **elements** that tie according to this criterion, the elements should be further sorted by some secondary key. These two levels of sorting are referred to as the primary and secondary sort respectively. Both sorts can be accomplished at once by an appropriately designed `compareTo()` **method**.

primitive data type A set of values together with operations that can be performed on them. The primitive data types in Java provide a set of basic building blocks from which all the more complex types of data can be built. There are three categories of primitive data type: numbers, characters and Booleans.

`private` In the case of **instance methods** and **instance variables**, a Java **access modifier** restricting access to instance variables or methods to instances of the **class** that declares them.

problem domain The collection of real-world entities within the application area that exhibit the behaviours that the required **software** has to model.

program **Software** that has a starting point at which it takes some input, after which it performs whatever computation is needed, and has an end point at which output is given and the software ceases to run.

programmer A developer whose role is to **implement** the code.

project documentation A written description of the activities, decisions and outcomes of a project's **phases**.

project failure A situation where a project fails to deliver the client's requirements.

project manager A person who plans and oversees the running of a **software development** project.

protocol The set of **messages** an **object** can respond to (understands).

prototype An early working version of the **software** or part of it.

pseudo-variable A special undeclared **variable**, **visible** within a **method** or **constructor**, that cannot be changed by **assignment**. Java has two such variables – `this` and `super`.

`public` In the case of **instance methods** and **instance variables**, a Java **access modifier** which allows access to instance variables or methods from instances of any other class.

Q

qualified When accessing a **class variable**, it is generally necessary to access it via the name of the **class** using the dot notation (e.g. `MyClass.myStaticVariable`). This is described as using the **qualified name** of the class variable. This is not strictly necessary within **class methods**, **instance methods** or **constructors** of the class that defines the class variable, but it is good programming style and complements the use of `this` for accessing **instance variables**. **Objects** of other classes that might want to access an accessible class variable *must* qualify the **variable** with the class name.

R

`Reader` The base class of a group of **classes** including `FileReader` and `BufferedReader`, which are used to read 16-bit character streams. `Reader` classes are used to read text files

receiver The **object** to which a **message** is sent.

recursive method A **method** which calls itself as part of its method definition. This can lead to indefinite looping when an attempt is made to execute the method.

refactoring A technique whereby code is rewritten, without changing its overall effect, for the purpose of improving its **design** by removing code duplication. The term can be applied to big changes to code whereby a whole **class** hierarchy is altered, for example by the introduction of an **abstract class**, or to relatively small changes to a single class by factoring out duplicated code that appears in the class's methods into other **helper methods**.

reference semantics The situation whereby a **variable** holds the *address* of an **object**, rather than a *value*. A **reference type variable** on the left-hand side of an **assignment statement** always ends up referring to the object on the right-hand side *(*cf. **value semantics**).

reference type variable A **variable** declared to reference an **object** of the declared or compatible type.

regression testing Testing that modifications or additions to one part of the code have not made any other part stop working properly.

requirements What is required of the **software**.

requirements specification Eliciting and analysing what the client wants in order to produce a detailed and complete specification of the **requirements** of the **software** in terms of what it should do.

review A point within an iterative **software development method** where developers and clients can take changes into account.

run-time Refers to the moment when a **program** begins to execute, in contrast to the time at which it has been loaded or compiled. The amount of time, elapsed time, used in executing a program is sometimes called the run-time.

run-time error A programming error that only becomes apparent when the **program** is run, and cannot be detected beforehand. Run-time errors can be **logical errors** or **exceptions**.

run-time system The Java **run-time system** consists of the **virtual machine** plus additional code that the **compiler** produces that has not been explicitly written into the **source code** by the programmer, but which is needed to allow the compiled code to run on the virtual machine.

S

`Scanner` A **class** used to read string tokens from a source. In M255 the source may be a `FileReader` or a `String`.

scope The scope of a **variable** describes the areas of program code from which the variable may be used. The scope of a local variable is the **statement block** in which it is declared (and any nested statement block). In discussing methods, the scope of a local variable is the method in which it is declared.

semantic error A semantic error arises from a misunderstanding of the meaning or effect of some construct in a programming language. Many such errors are detected during compilation by the **parser**.

semantics That part of the definition of a language concerned with specifying the meaning or effect of text that is constructed according the **syntax** rules of the language.

sequence diagram A diagram that depicts the interactions required between **objects** to carry out a particular task; this collaboration is shown in the form of **messages** and **message answers**.

serialisation The process by which an **object** implementing the `Serializable` **interface** can be written to an `ObjectOutputStream` as a sequence of bytes.

set An unordered **collection** which has no **indexing**, that contains no duplicates and which has a dynamic size (i.e. it can grow and shrink as **elements** are added and removed).

setter message A message that sets the value of one of a **receiver's attributes**. See **getter message** and **accessor message**.

setter method An **accessor method** whose purpose is to assign a new value to an instance variable. The new value is determined by the single **argument** of the **method**.

signature See **method signature**.

software A general term for all the **applications**, **programs** and **systems** that run on your computer.

software component A reusable piece of software, with a well-documented interface, that can be combined with other components to construct various software solutions.

software developer An umbrella title, referring to someone who takes on one or more of a range of jobs within **software development**.

software development A planned, phased process, involving modelling different aspects of the **software** as well as implementing, **testing** and maintaining it.

software development method A particular set of development **phases** and activities, applied in a particular order.

software engineering A term used to refer to a wide range of concerns connected with carrying out systematic **software development**.

software model An illustration or description of the **software**, or of part of it, which emphasises certain aspects and omits others.

software pattern A tried and tested solution to a programming problem that comes up again and again, presented as a pattern of **classes** and **collaborations**.

software tester A developer whose role is to test the **software** as it is being developed.

sorted collection A sorted collection is a collection that always keeps its **elements** in their **natural ordering**. This contrasts with an **ordered collection**, where the ordering may just be an accident of where elements happen to have been placed in the collection.

source code Code expressed in a **high-level** programming language. The term is also often applied to code that does not fully conform to the language (it contains an error), but with minor correction would conform.

stable sort A sort that does not reorder equal **elements** is known as a **stable sort**.

standard default value The default values provided by Java to newly created **variables**.

state The values of the **attributes** of an **object** constitute its state. The state of an object can vary over time as the values of its attributes change.

statement A statement represents a single instruction for the **compiler** or interpreter to translate into **bytecode**. In Java a statement must always end with a semicolon.

statement block A statement or sequence of **statements** 'bundled together' for use in a particular context. Any sequence of statements can be turned into a block by enclosing it in braces.

`static` A Java keyword which defines a **variable** or **method** as belonging to a **class** rather than its **instances**.

static model An illustration of the state of the **software**, or part of it, at a particular imagined time during execution.

stream An **object** used to connect a data source to a data sink, enabling data to be transferred from the source to the sink.

strict UML diagram A diagram that adheres strictly to the **UML standard**.

`String` A **class** whose **instances** represent a sequence of characters. Such objects are **immutable**.

`StringBuilder` A **class** whose **instances** represent a sequence of characters. Such **objects** are mutable.

sub-array A part of an **array** consisting of contiguous components.

subclass A subclass is any **class** which, when taking part in an **inheritance** relationship with another class, is the class to inherit functionality. In Java all classes except `Object` are subclasses of some other class.

subclassing A technique for defining a **class** as a **subclass** of an existing class.

subinterface A subinterface is an **interface** that contains all the **signatures** of some parent interface (called its **superinterface**), and which typically contains additional signatures.

sublist A **list**, together with any two positions in the list may be seen as defining a new list containing just those **elements** between the two positions, retaining the same order. The new list is known as a sublist.

substitutability The principle in object-oriented programming whereby, wherever the system expects an **object** of type X, an object of **class** Y can always be substituted instead, where class Y is a **subclass** of X, or where class Y implements an **interface** of type X. See **substitution**.

substitution The technique of providing an **instance** of one **class** in a situation where an instance of a different class is expected. In Java, an object can be **assigned** to a **variable** whose type has been declared as some **superclass** of the **object**, or which has been declared as an **interface** type which that object's class implements. Similarly, an object can be used as an actual **argument** to a **method** where the **formal argument's** type has been declared as some superclass of the object, or which has been declared as an interface type which that object's class implements.

`super` The pseudo-variable `super` is used within a **method** to refer to the **receiver**, so that an **object** can send a **message** to itself. However, while the use of `this` causes the JVM to start its search for the corresponding method in the **class** of the receiver, the use of `super` causes the JVM to start its search for the method in the **superclass** of the class containing the method in which `super` appears.

`super()` Used within a **constructor** to invoke the constructor without any **arguments** in the **direct superclass**.

superclass If B is a **subclass** of A, then A is the superclass of B.

superinterface A superinterface is an **interface** viewed from the perspective of another interface (called its **subinterface**). The subinterface contains all the **signatures** of the superinterface and typically contains additional signatures.

syntax Rules defining the legal sequences of characters in a programming language. Syntax rules define the form of the various constructs in the language, but say nothing about the meaning of these constructs.

syntax error Failure to observe a **syntax** rule. Such an error is detected during compilation by the **parser**.

system **Software** that is intended to run forever, responding to events in often complex ways.

system testing Testing that the overall system functions correctly.

systems analyst A developer who ideally has knowledge of both the **problem domain** (the client's business needs) and software design and whose role is to analyse the feasibility of proposed **software** and how it will impact on the client's business practices.

T

technical writer A developer whose role is to develop user documentation.

test data A set of data used as the inputs to a test.

test-driven development An approach to **software development** in which code is written in increments, with the tests for each increment being written before the code.

test driver tool A **testing tool** that executes the **software** being tested with specified inputs.

test-first A **software development** philosophy in which tests are written first and the corresponding code follows.

test harness A programme written solely to test other **software components**.

testing The process of uncovering faults in **software**.

testing framework An API which provides support for automated software testing, allowing tests to be created in a standardised way and reused.

testing tool A **CASE tool** that aids some aspect of **testing**.

`this` A pseudo-variable which, in a **method** or **constructor**, acts as a reference to the **object** which is executing that method or constructor, so that the object can send a **message** to itself.

throw the process of throwing an exception, see **exception**.

`try`–`catch` statement An exception handler, a mechanism to catch **exceptions**.

two-dimensional array An **array** whose **components** hold **arrays**.

U

UML (Unified Modeling Language) A **modelling language** based on diagrams.

UML standard The currently accepted specification of what is valid **UML** and how it should be used.

UML-type diagram A diagram that varies in some small way from the strict **UML standard**.

unchecked exception An **exception** that the **compiler** *does not* require the programmer to handle if they write code that executes a **method** that can throw such an exception.

union Union is a mathematical operation on two or more **sets** that results in a set containing all the **elements** of all the sets. In Java, union is implemented by an `addAll()` message. Given two sets, `set1` and `set2`, the message-send `set1.addAll(set2)` would add to `set1` all the elements contained by `set2`. The term intersection is used to refer both to the result and to the operation.

unit testing A test of a single **method**.

V

value semantics The situation in which a **variable** holds a value, of some **primitive data type**. A **value type variable** on the left-hand side of an **assignment statement** always ends up holding a copy of the value on the right-hand side (cf. **reference semantics**).

value type variable A **variable** declared to hold a value of the declared or compatible **primitive** type.

variable A named 'chunk' or block of the computer's memory which can hold either a value of some **primitive** type or the address (reference) of an **object**.

variable reference diagram A diagram showing a **reference type variable** pointing to a representation of an **object** with the current values of its **attributes**.

virtual machine A layer of **software** that simulates a computer capable of interpreting **bytecode**.

visibility An **object** may contain an **inherited instance variable**, but be unable to access it directly because it was declared `private` in a **superclass**. However, a `public` **accessor method** declared in the superclass may allow the object to access this variable indirectly. In such a situation, the variable is said to lack **visibility**.

visual representation A useful representation of a software **object** which, by definition, is invisible. The visual representation may be textual, such as characters in a word processor; or graphical, such as shapes in a drawing application.

W

waterfall method A traditional and idealised view of developing software by strictly following a sequence of **phases**.

white box testing Testing which is guided by information about internal structure and which aims to ensure that every part of that structure has been tried.

workspace variable A Java development environment that provides a workspace, like the one in M255, will have special **workspace variables**. Workspace variables behave like **local variables**, except that their lifetime lasts until you close or reset the workspace, rather than ending when a particular block has finished executing.

wrapper class See **autoboxing** to understand why these **classes** need to exist. Every **primitive** type has a corresponding wrapper class whose purpose is to hold a primitive value in a place where a primitive value is intuitively what is needed, but where the type checking requires an **object**. For example, the wrapper class of `int` is `Integer`.

`Writer` The base **class** of a group of classes including `FileWriter` and `BufferedWriter`, which are used to write 16-bit character streams. `Writer` classes are used to write text files.