Glossary of some OOP Terms

abstract class

A class with the abstract reserved word in its header. Abstract classes are distinguished by the fact that you may not *directly* construct objects from them using the new operator. An abstract class may have zero or more *abstract methods*.

abstraction

A simplified representation of something that is potentially quite complex. It is often not necessary to know the exact details of how something works, is represented or is implemented, because we can still make use of it in its simplified form. Object-oriented design often involves finding the right level of abstraction at which to work when modeling real-life objects. If the level is too high, then not enough detail will be captured. If the level is too low, then a program could be more complex and difficult to create and understand than it needs to be.

abstract method

A method with the abstract reserved word in its header. An abstract method has no *method body*. Methods defined in an *interface* are always abstract. The body of an abstract method must be defined in a *sub class* of an *abstract class*, or the body of a class implementing an interface.

accessor method

A method specifically designed to provide access to a private attribute of a class. By convention, we name accessors with a get prefix followed by the name of the attribute being accessed. For instance, the accessor for an attribute named speed would be getSpeed. By making an attribute private, we prevent objects of other classes from altering its value other than through a *mutator method*. Accessors are used both to grant safe access to the value of a private attribute and to protect attributes from inspection by objects of other classes. The latter goal is achieved by choosing an appropriate visibility for the accessor.

aggregation

A relationship in which an object contains one or more other subordinate objects as part of its state. The subordinate objects typically have no independent existence separate from their containing object. When the containing object has no further useful existence, neither do the subordinate objects. For instance, a gas station object might contain several pump objects. These pumps will only exist as long as the station does. Aggregation is also referred to as the *has-a relationship*, to distinguish it from the *is-a relationship*, which refers to *inheritance*.

anonymous class

A class created without a class name. Such a class will be an *sub class* or an implementation of an *interface*, and is usually created as an *actual argument* or returned as a method result. For instance

quitButton.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){

System.exit(0);

}

});

argument

Information passed to a *method*. A method expecting to receive arguments must contain a *formal argument* (parameter) declaration for each as part of its *method header*. When a method is called, the *actual argument* values are copied into the corresponding formal arguments (parameter).

attribute

A particular usage of an *instance variable*. The set of attribute values held in a particular *instance* of a class define the current *state* of that instance. A class definition may impose particular constraints on the valid states of its instances by requiring that a particular attribute, or set of attributes, do not take on particular values. For instance, attributes holding coursework marks for a class should not hold negative values. Attributes should be manipulated by *accessor* and *mutator* methods.

bytecode

Java source files are translated by a *compiler* into bytecodes - the *instruction set* of the *Java Virtual Machine (JVM)*. Bytecodes are stored in .class files.

class

A programming language concept that allows data and *methods* to be grouped together. The class concept is fundamental to the notion of an *object-oriented programming language*. The methods of a class define the set of permitted operations on the class's data (its *attributes*). This close tie between data and operations means that an *instance* of a class - an *object* - is responsible for responding to messages received via its defining class's methods.

class header

The header of a *class* definition. The header gives a name to the class and defines its *access*. It also describes whether the class extends a *super class* or implements any *interfaces*.

class inheritance

When a *super class* is extended by a *sub class*, a class inheritance relationship exists between them. The sub class inherits the methods and attributes of its super class. In Java, class inheritance is *single inheritance*. See *interface inheritance* for an alternative form of inheritance.

class method

A synonym for *static method*.

class scope

Private *variables* defined outside the *methods* within a class have class scope. They are accessible from all methods within the class, regardless of the order in which they are defined. Private methods also have class scope. Variables and methods may have a wider *scope* if they do not use the private access modifier.

class variable

A synonym for *static variable*.

‘True’ encapsulation

Safeguarding the state of an objects by defining its attributes as private and channeling access to them through *accessor* and *mutator* methods.

Encapsulation

Encapsulation is the packing of data/attributes and methods/functions into a single component. The features of encapsulation are supported using classes in most object-oriented programming languages. It allows selective **hiding** of properties and methods in an object by building an impenetrable wall (using scope) to protect the code from accidental corruption.

Graphical User Interface

A Graphical User Interface (GUI) is part of a program that allows user interaction via graphical components, such as menus, buttons, text areas, etc. Interaction often involves use of a mouse.

information hiding

The practice of ensuring that only as much information is revealed about the implementation of a class as is strictly required. Hiding unnecessary knowledge of implementation makes it less likely that other classes will rely on that knowledge for their own implementation. This tends to reduce the strength of *coupling* between classes. It also reduces that chance that a change of the underlying implementation will break another class. Ensuring that all *fields* of a class are defined as private, is one of the ways that we seek to promote information hiding.

inheritance

A feature of *object-oriented programming languages* in which a *sub type* inherits *methods* and *variables* from its *super type*. Inheritance is most commonly used as a synonym for *class inheritance* {class!inheritance}, but *interface inheritance* is also a feature of some languages, including Java.

inheritance hierarchy

The relationship between *super classes* and *sub classes* is known as an inheritance hierarchy. *Single inheritance* of classes means that each class has only a single `parent' class and that the Object class is the ultimate ancestor of all classes - at the top of the hierarchy. Two classes that have the same immediate super class can be thought of as *sibling sub classes*. *Multiple inheritance* of interfaces gives the hierarchy a more complex structure than that resulting from simple *class inheritance*.

inner class

A class defined inside an enclosing class or method. We use the term to refer to non-static *nested classes*.

instance

A synonym for *object*. Objects of a class are *instantiated* when a class *constructor* is invoked via the *new operator*.

instance variable

A non-static *field* of a *class*. Each individual object of a class has its own copy of such a field. This is in contrast to a *class variable* which is shared by all instances of the class. Instance variables are used to model the *attributes* of a class.

instantiation

The creation of an *instance* of a class - that is, an *object*.

interface inheritance

When a *class* implements an *interface*, an interface *inheritance* relationship exists between them. The class inherits no implementation from the interface, only method signatures and *static variables*. It is also possible for one interface to extend one or more interfaces. In Java, interface inheritance is the only form of *multiple inheritance*. See *class inheritance* for an alternative form of inheritance.

Java Virtual Machine (JVM)

An idealized machine whose *instruction set* consists of *bytecodes*. A Java program is *compiled* to an equivalent bytecode form and executed on an *interpreter* which implements the JVM.

layout manager

An object responsible for sharing the available space between multiple components within a graphical container.

local variable

A variable defined inside a *method body*.

main method

The starting point for program execution

public static void main(String[] args)

method

The part of a *class definition* that implements some of the behavior of objects of the class. The body of the method contains *declarations* of *local variables* and *statements* to implement the behavior. A method receives input via its *arguments*, if any, and may return a result if it has not been declared as void.

method body

The body of a method: everything inside the outermost *block* of a method.

method header

The header of a method, consisting of the method name, its result type, formal arguments and any exceptions thrown. Also known as a *method signature*.

method overloading

Two or more methods with the same name defined within a class are said to be overloaded. This applies to both constructors and other methods. Overloading applies through a class hierarchy, so a sub class might overload a method defined in one of its super classes. It is important to distinguish between an overloaded method and an *overridden method*. Overloaded methods must be distinguishable in some way from each other; either by having different numbers of arguments, or by the types of those arguments being different. Overridden methods have identical formal arguments.

method overriding

A method defined in a *super class* may be overridden by a method of the same name defined in a *sub class*. The two methods must have the same name and number and types of formal arguments. Any checked exception thrown by the sub class version must match the type of one thrown by the super class version, or be a sub class of such an exception. However, the sub class version does not have to throw any exceptions that are thrown by the super class version. It is important to distinguish between method overriding and *method overloading*. Overloaded methods have the same names, but differ in their formal arguments.

multiple inheritance

The ability of a class or interface to extend more than one class or interface. In Java, multiple inheritance is only available in the following circumstances

* An interface may extend more than one interface.
* A class may implement more than one interface.

Only *single inheritance* is possible for a class extending another class.

no-arg constructor

A constructor that takes no arguments. By default, all classes without an explicit constructor have a default no-arg constructor with public access. Its role is purely to invoke the no-arg constructor of the immediate super class.

object

An *instance* of a particular *class*. In general, any number of objects may be constructed from a class definition (see *singleton*, however). The class to which an object belongs defines the general characteristics of all instances of that class. Within those characteristics, an object will behave according to the current state of its *attributes* and environment.

package

A named grouping of classes and interfaces that provides a package *namespace*. Classes, interfaces and class members without an explicit public, protected or private *access modifier* {access!modifier} have *package visibility*. Public classes and interfaces may be imported into other packages via an *import statement*.

primitive type

Java's eight standard non-class types are primitive types: boolean, byte, char, double, float, int, long and short.

return type

The declared type of a method, appearing immediately before the method name, such as void in

public static void main(String[] args)

or Point[] in

public Point[] getPoints()

scope

A language's scope rules determine how widely variables, methods and classes are visible within a class or program. Local variables have a scope limited to the *block* in which they are defined, for instance. Private methods and variables have *class scope*, limiting their accessibility to their defining class. Java provides private, package, protected and public visibility.

static method

A static method (also known as a *class method*) is one with the static reserved word in its header. Static methods differ from all other methods in that they are not associated with any particular instance of the class to which they belong. They are usually accessed directly via the name of the class in which they are defined.

static variable

A static variable defined inside a *class body*. Such a variable belongs to the class as a whole, and is, therefore, shared by all objects of the class. A class variable might be used to define the default value of an *instance variable*, for example, and would probably also be defined as final, too. They are also used to contain dynamic information that is shared between all instances of a class. For instance the next account number to be allocated in a bank account class. Care must be taken to ensure that access to shared information, such as this, is synchronized where multiple threads could be involved. Class variables are also used to give names to application-wide values or objects since they may be accessed directly via their containing class name rather than an instance of the class.

sub class

A class that extends its *super class*. A sub class *inherits* all of the members of its super class. All Java classes are sub classes of the Object class, which is at the root of the *inheritance hierarchy*.

super class

A class that is extended by one or more *sub classes*. All Java classes have the Object class as a super class.

this

A Java reserved word with several different uses:

* Within a constructor, it may be used as the first statement to call another constructor in the same class. For example
* // Initialise with default values.
* public Heater()
* {
* // Use the other constructor.
* this(15, 20);
* }
* // Initialise with the given values.
* public Heater(int min,int max)
* {
* ...
* }

* Within a constructor or method, it may be used to distinguish between a field and a parameter or method variable of the same name. For instance:
* public Heater(int min,int max)
* {
* this.min = min;
* this.max = max;
* ...
* }

* It can be used as a reference to the current object, typically in order to pass a reference to another object:
* talker.talkToMe(this);

wrapper classes

Java's *primitive types* are not object types. The wrapper classes are defined in the java.lang *package*. They consist of a class for each primitive type: Boolean, Byte, Character, Double, Float, Integer, Long and Short. These classes provide methods to parse strings containing primitive values, and turn primitive values into strings. The Double and Float classes also provide methods to detect special bit patterns for floating point numbers, representing values such as NaN, +infinity and -infinity.