A

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.

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.

actual argument *or* actual parameter

The value of an argument passed to a method from outside the method. When a method is called, the *actual argument* values are copied into the corresponding *formal arguments*. The types of the actual arguments must be compatible with those of the formal arguments.

argument

Information passed to a *method*. Arguments are also sometimes called parameters. A method expecting to receive arguments must contain a *formal argument* 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.

arithmetic expression

An expression involving numerical values of integer or floating point types. For instance, operators such as `+`, `-`, `*`, `/` and `%` take arithmetic expressions as their operands and produce arithmetic values as their results.

arithmetic operator

Operators, such as `+`, `-`, `*`, `/` and `%`, that produce a numerical result, as part of an *arithmetic expression*.

array

A fixed-size object that can hold zero or more items of the array's declared type.

array initializer

An initializer for an array. The initializer takes the place of separate creation and initialization steps. For instance, the initializer

```
int[] pair = { 4, 2, };
```

is equivalent to the following four statements.

```
int[] pair;
pair = new int[2];
pair[0] = 4;
pair[1] = 2;
```

assignment operator

The operator (=) used to store the value of an expression into a *variable*, for instance

```
variable = expression;
```

The right-hand-side is completely evaluated before the assignment is made. An assignment may, itself, be used as part of an expression. The following *assignment statement* stores zero into both variables.

```
x = y = 0;
```

assignment statement

A statement using the *assignment operator*.

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.

B

binary

Number representation in base 2. In base 2, only the digits `0` and `1` are used. Digit positions represent successive powers of 2. See *bit*.

binary operator

An *operator* taking two operands. Java has many binary operators, such as the arithmetic operators `+`, `-`, `*`, `/` and `%`, and the boolean operators `&&`, `||` and `^`, amongst others.

bit

A binary digit, which can take on two possible values: `0` and `1`. Bits are the fundamental building block of both programs and data. Computers regularly move data around in multiples of eight-bit units (*bytes* for the sake of efficiency).

block

Statements and declarations enclosed between a matching pair of curly brackets (`{` and `}`). For instance, a *class body* is a block, as is a *method body*. A block encloses a nested *scope* level.

boolean

One of Java's *primitive types*. The `boolean` type has only two values: `true` and `false`.

boolean expression

An expression whose result is of type `boolean`, i.e. gives a value of either `true` or `false`. Operators such as `&&` and `||` take boolean operands and produce a boolean result. The relational operators take operands different types and produce boolean results.

bounds

The limits of an *array* or collection. In Java, the lower limit is always zero. In the case of an array, the upper bound is one less than then length of the array, and is fixed. Indexing outside the bounds of an array or collection will result in an `IndexOutOfBoundsException` *exception* being thrown.

---

break statement
: A statement used to break out of a loop, switch statement or labeled block. In all cases, control continues with the statement immediately following the containing block.

byte
: In general computing, this refers to eight *bits* of data. In Java it is also the name of one of the primitive data types, who size is eight bits.

C

case label
: The value used to select a particular case in a *switch statement*.

cast
: Where Java does not permit the use of a source value of one type, it is necessary to use a cast to force the compiler to accept the use for the target type. Care should be taken with casting values of primitive types, because this often involves loss of information. Casts on object references are checked at runtime for legality. A `ClassCastException` *exception* will be thrown for illegal ones.

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 body
: The body of a *class* definition. The body groups the definitions of a class's *members* - *fields*, *methods* and *nested classes*.

class constant
: A variable defined as both `final` and `static`.

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*.

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*.

cohesion
The extent to which a component performs a single well-defined task. A strongly cohesive method, for instance, will perform a single task, such as adding an item to a data structure, or sorting some data, whereas a weakly cohesive method will be responsible for several disparate tasks. Weakly cohesive components should, in general, be split into separate more cohesive components. The java.util package is a weakly cohesive package because it contains many unrelated classes and interfaces, whereas the java.io package is highly cohesive.

comment
A piece of text intended for the human reader of a program. Compilers ignore their contents.

compilation
The process of translating a programming language. This often involves translating a *high level programming language* into a *low level programming language*, or the binary form of a particular *instruction set*. The translation is performed by a program called a *compiler*. A Java compiler translates programs into *bytecodes*.

compiler
A program which performs a process of *compilation* on a program written in a *high level programming language*.

condition
A *boolean expression* controlling a conditional statement or loop.

conditional operator
An *operator* taking three operands - a ternary operator. The conditional operator (`?:`) is used in the form

```
bexpr ? expr1 : expr2
```

where `bexpr` is a *boolean expression*. The boolean expression has the value `true` then the result of the operation is the value of `expr1`, otherwise it is the value of `expr2`.

constant
A *variable* whose value may not be changed. In Java, these are implemented by *final variables*.

constructor
A constructor is automatically called when an *instance* of its class is created. A constructor always has the same name as its class, and has no return type. For instance

```
public class Ship {
    public Ship(String name){
        ...
    }
    ...
}
```

A class with no explicit constructor has an implicit *no-arg constructor*, which takes no arguments and has an empty body.

continue statement
A statement that may only be used inside the body of a loop. In the case of a while loop or do loop, control passes immediately to the loop's terminating test. In the case of a for loop, control passes to the post-body update expression.

---

control structure
A statement that affects the *flow of control* within a method. Typical control structures are loops and if statements.

D

decrement operator
An operator (`--`) that adds one to its operand. It has two forms: pre-decrement (`--x`) and post-decrement (`x--`). In its pre-decrement form, the result of the expression is the value of its argument *after* the decrement. In its post-decrement form, the result is the value of its argument *before* the decrement is performed. After the following,

```
int a = 5, b = 5;
int y,z;
y = --a;
z = b--
```

`y` has the value `4` and `z` has the value `5`. Both `a` and `b` have the value `4`.

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

data type
There are eight primitive data types in Java; five of these represent numerical types of varying range and precision - `double`, `float`, `int`, `long` and `short`. The remaining three are used to representing single-bit values (`boolean`), single byte values (`byte`) and two-byte characters from the ISO Unicode character set (`char`).

decimal
Number representation in base 10. In base 10, the digits `0` to `9` are used. Digit positions represent successive powers of 10.

default initial value
The default value of any variable not explicitly initialized when it is declared. Fields of numeric primitive types have the value zero by default, `boolean` variables have the value `false`, `char` variables have the value `\u0000` and object references have the value `null`. The initial values of local variables are undefined, unless explicitly initialized.

default label
The destination for all values used in a *switch statement* expression that do not have explicit *case labels*. A default label is optional.

do loop
One of Java's three *control structures* used for looping. The other two are the *while loop* and *for loop*. A do loop consists of a loop body and a *boolean expression*. The condition is tested after the loop body has been completed for the first time and re-tested each time the end of the body is completed. The loop terminates when the condition gives the value `false`. The statements in the loop body will always be executed at least once.

E

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

enum
: An enum type is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.
Because they are constants, the names of an enum type's fields are in uppercase letters. In the Java programming language, you define an enum type by using the enum keyword. For example, you would specify a days-of-the-week enum type as:

```
public enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
   THURSDAY, FRIDAY, SATURDAY
}
```

exclusive-or operator
: The exclusive-or *boolean* operator (`^`) gives the value `true` if only one of its operands is `true`, otherwise it gives the value `false`. Similarly, the bit manipulation version produces a `1` bit wherever the corresponding bits in its operands are different.

expression
: A combination of *operands* and *operators* that produces a resulting value. Expressions have a resulting type, that affects the context in which they may be used. See *boolean expression* and *arithmetic expression*, for instance.

F

field
: *Variables* defined inside a class or interface, outside of the methods. Fields are *members* of a class.

final variable
: A variable with the `final` reserved word in its declaration. A final may not assigned to once it has been initialized. Initialization often takes place as part of its declaration. However, the initialization of an uninitialized final *field* (known as a *blank final variable*) may be deferred to the class's *constructor*, or an *initializer*.

first in, first out
: The (FIFO) semantics of a *queue* data structure. Items are removed in the order in which they arrived in the queue, so older items are always removed before newer ones. See *last in, first out*.

floating point number
: See *real number*.

for loop
: One of Java's three *control structures* used for looping. The other two are the *while loop* and *do loop*. A for loop consists of a loop header and a loop body. The header consists of three expressions separated by two semicolons and one or more of these may be omitted. The first expression is only evaluated once, at the point the loop is

entered. The middle expression is a *boolean expression* representing the loop's termination test. An empty expression represents the value `true`. The third expression is evaluated after each completion of the loop's body. The loop terminates when the termination test gives the value `false`. The statements in the loop body might be executed zero or more times.

formal argument or formal parameter

The definition of a method's argument which are part of a *method header*. Each formal argument has an associated type. When a method is called, the *actual argument* values are copied into the corresponding formal arguments. The types of the actual arguments must be compatible with those of the formal arguments.

H

hash code

A value returned by a *hash function*. A hash code can be used as an index into a random-access data structure, providing an efficient mapping between an object and its location. Used by classes such as `HashMap`.

hash function

A function used to produce a *hash code* from the arbitrary contents of an object. Classes can override the `hashValue` method, inherited from the `Object` class, to define their own hash function.

heterogeneous collection

A collection of objects with different *dynamic types*. See *homogeneous collection*.

hexadecimal

Number representation in base 16. In base 16, the digits `0` to `9` and the letters `A` to `F` are used. `A` represents 10 (base 10), `B` represents 11 (base 10), and so on. Digit positions represent successive powers of 16.

homogeneous collection

A collection of objects with the same *dynamic type*. Arrays are the most common homogeneous collection objects. See *heterogeneous collection*.

I

identifier

A programmer-defined name for a *variable*, *method*, *class* or *interface*.

if-else statement

A *control structure* used to choose between performing one of two alternative actions.

```
if(boolean-expression){
    // Statements performed if expression is true.
    ...
}
else{
    // Statements performed if expression is false.
    ...
}
```

It is controlled by a *boolean expression*. See *if statement*.

if statement

A *control structure* used to choose between performing or not performing further actions.

```
if(boolean-expression){
    // Statements performed if expression is true.
    ...
}
```

It is controlled by a *boolean expression*. See *if-else statement*.

implicit type conversion

Type conversion that does not require a *cast*. Implicit type conversions typically do not involve any loss of information. For instance, combining an integer operand with a floating point operand in an *arithmetic expression* will result in an implicit type conversion of the integer to an equivalent floating point value.

import statement

A statement that makes the names of one or more classes or interfaces available in a different *package* from the one in which they are defined. Import statements follow any *package declaration* {package!declaration}, and precede any class or interface definitions.

increment operator

An operator (`++`) that adds one to its operand. It has two forms: pre-increment (`++x`) and post-increment (`x++`). In its pre-increment form, the result of the expression is the value of its argument *after* the increment. In its post-increment form, the result is the value of its argument *before* the increment is performed. After the following,

```
int a = 5, b = 5;
int y,z;
y = ++a;
z = b++
```

`y` has the value `6` and `z` has the value `5`. Both `a` and `b` have the value `6`.

infinite loop

A loop whose termination test never evaluates to `false`. Sometimes this is a deliberate act on the part of the programmer, using a construct such as

```
while(true) ...
```

or

```
for( ; ; ) ...
```

but it can sometimes be the result of a logical error in the programming of a normal loop condition or the statements in the body of the loop.

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*.

initializer
: A *block* defined at the outermost level of a class - similar to a method without a header. Initializer blocks are executed, in order, when an *instance* is created. They are executed before the *constructor* of the defining class, but after any *super class* constructor. They are one of the places in which *blank final variables* may be initialized.

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*.

integer
: A positive or negative whole number. The *primitive types* `byte`, `short`, `int` and `long` are used to hold integer values within narrower or wider ranges.

interpreter
: A program which executes a translated version of a source program by implementing a *virtual machine*. Interpreters typically simulate the actions of an idealized *Central Processing Unit*. An interpreter for Java must implement the *Java Virtual Machine (JVM)* and executes the *bytecodes* produced by a Java *compiler*. The advantage of using an interpreter for Java is that it make the language more *portable* than if it were a fully compiled language. The bytecode version of a program produced by a Java compiler may be run on any interpreter implementing the JVM.

is-a relationship
: See *inheritance*.

iteration
: Repetition of a set of statements, usually using a looping *control structure*, such as a *while loop*, *for loop* or *do loop*.

K

key value
: The object used to generate an associated *hash code* for lookup in an associative data structure.

L

last in, first out
: The (LIFO) semantics of a *stack* data structure. Items are removed in the opposite order to which they arrived in the stack, so newer items are always removed before older ones. See *first in, first out*.

---

local variable
A variable defined inside a *method body*.

logical error
An error in the logical of a method or class. Such an error might not lead to an immediate *runtime error*, but could have a significant impact on overall program correctness.

logical operators
Operators, such as `&&`, `||`, `&`, `|` and `^` that take two boolean operands and produce a boolean result. Used as part of a *boolean expression*, often in the condition of a *control structure*.

loop variable
A *variable* used to control the operation of a loop, such as a *for loop*. Typically, a loop variable will be given an initial value and it is then incremented after each *iteration* until it reaches or passes a terminating value.

M

main method
The starting point for program execution
`public static void main(String[] args)`

member
For now, the members of a class are *fields* and *methods*.

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.

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. See *overriding for breadth*, *overriding for chaining* and *overriding for restriction*.

method result
: The value returned from a method via a *return statement*. The type of the expression in the return statement must match the return type declared in the *method header*.

method signature
: The signature of a method consists of its name and the types of its parameters (enclosed in parentheses and separated by commas).

method variable
: See *local variable*.

mutator method
: A method specifically designed to allow controlled modification of a `private` attribute of a class. By convention, we name mutators with a `set` prefix followed by the name of the attribute being modified. For instance, the mutator for an attribute named `speed` would be `setSpeed`. By making an attribute private, we prevent objects of other classes from altering its value other than through its mutator. The mutator is able to check the value being used to modify the attribute and reject the modification if necessary. In addition, modification of one attribute might require others to be modified in order to keep the object in a consistent state. A mutator method can undertake this role. Mutators are used both to grant safe access to the value of a private attribute and to protect attributes from modification by objects of other classes. The latter goal is achieved by choosing an appropriate visibility for the mutator.

N

newline
: The `\n` character.

new operator
: The operator used to create *instances* {instance} of a class.

null character
: The `\u0000` character. Care should be taken not to confuse this with the *`null` reference*.

null reference
: A value used to mean, `no object'. Used when an object reference variable is not referring to an object.

O

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.

object construction
: The creation of an `object`, usually via the `new` operator. When an object is created, an appropriate *constructor* from its class is invoked.

---

object-oriented language
: Programming languages such as C++ and Java that allow the solution to a problem to be expressed in terms of objects which belong to classes.

object reference
: A reference to an object. Languages other than Java use term's such as *address* or *pointer*. It is important to keep the distinction clear between an object and its reference. A variable such as `argo`
```
Ship argo;
```
is capable of holding an object reference, but is not, itself, an object. It can refer to only a single object at a time, but it is able to hold different object references from time to time.

octal
: Number representation in base 8. In base 8, only the digits `0` to `7` are used. Digit positions represent successive powers of 8.

operand
: An operand is an argument of an *operator*. Expressions involve combinations of operators and operands. The value of an expression is determined by applying the operation defined by each operator to the value of its operands.

operator
: A symbol, such as `-`, `==` or `?:` taking one, two or three operands and yielding a result. Operators are used in both *arithmetic expressions* and *boolean expressions*.

operator precedence
: See *precedence rules*

out of scope
: A *variable* is in *scope* as long as the program's *flow of control* is within the variable's defining *block*. Otherwise, it is out of scope.

P

parameter
: See *argument*.

polymorphism
: The ability of an object reference to be used as if it referred to an object with different forms. Polymorphism in Java results from both *class inheritance* and *interface inheritance*. The apparently different forms often result from the *static type* of the variable in which the reference is stored. Given the following class header
```
class Rectangle extends Polygon implements Comparable
```
an object whose *dynamic type* is Rectangle can behave as all of the following types: Rectangle, Polygon, Comparable, Object.

popup menu
: A menu of actions that is normally not visible on the screen until a mouse button is clicked. Popup menus help to keep a user interface from becoming cluttered.

post-decrement operator
: See *decrement operator*.

post-increment operator
: See *increment operator*.

---

precedence rules
The rules that determine the order of evaluation of an expression involving more than one *operator*. Operators of higher precedence are evaluated before those of lower precedence. For instance, in the expression `x+y*z`, the multiplication is performed before the addition because `*` has a higher precedence than `-`.

pre-decrement operator
See *decrement operator*.

pre-increment operator
See *increment operator*.

primitive type
Java's eight standard non-class types are primitive types: `boolean`, `byte`,

protected access
Protected access is available to a class *member* prefixed with the `protected` access modifier. Such a member is accessible to all classes defined within the enclosing *package*, and any *sub classes* extending the enclosing class.

public interface
The *members* of a class prefixed with the `public` access modifier. All such members are visible to every class within a program.

Q

queue
See *first in, first out (FIFO) queue*.

quotient
When integer division is performed, the result consists of a quotient and a remainder. The quotient represents the integer number of times that the divisor divides into the dividend. For instance, in `5/3`, `5` is the dividend and `3` is the divisor. This gives a quotient of `1` and a remainder of `2`.

R

real number
A number with an integer and a fractional part. The *primitive types* `double` and `float` are used to represent real numbers.

relational operators
Operators, such as `<`, `>`, `<=`, `>=`, `==` and `!=`, that produce a boolean result, as part of a *boolean expression*.
A relative filename could refer to different files at different times, depending upon the context in which it is being used. See *absolute filename*.

reserved word
A word reserved for a particular purpose in Java, such as `class`, `int`, `public`, etc. Such words may not be used as ordinary *identifiers*.

return statement
A statement used to terminate the execution of a method. A method with `void` *return type* may only have return statements of the following form
`return;`
A method with any other return type must have at least one return statement of the form

---

```
return expression;
```

where the type of `expression` must match the return type of the method.

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()
```

return value

The value of the *expression* used in a *return statement*.

runtime error

An error that causes a program to terminate when it is being run.

S

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.

semantic error

An error in the meaning of program. A statement may have no *syntax errors*, but might still break the rules of the Java language. For instance, if `ivar` is an `int` variable, the following statement is syntactically correct

```
ivar = true;
```

However, it is semantically incorrect, because it is illegal to assign a `boolean` value to an integer variable.

short-circuit operator

An *operator* in which only as many *operands* are evaluated as are needed to determine the final result of the operation. The logical-and (`&&`) and logical-or (`||`) operators are the most common example, although the *conditional operator* (`?:`) also only ever evaluates two of its three operands. See *fully evaluating operator*.

sibling sub classes

Classes that have the same immediate super class. See *inheritance hierarchy*.

single inheritance

In Java, a class may not extend more than one class. This means that Java has a single inheritance model for *class inheritance*. See *multiple inheritance* for the alternative.

single line comment

A *comment* in the form

```
// This line will be ignored by the compiler.
```

stack

See *last in, first out (LIFO) stack*.

state

Objects are said to possess state. The current state of an object is represented by the combined values of its attributes. Protecting the state of an object from inappropriate inspection or modification is an important aspect of class design and we recommend the use of *accessor methods* and *mutator methods* to facilitate attribute protection and integrity. The design of a class is often an attempt to model the states of objects in the

real-world. Unless there is a good match between the data types available in the language and the states to be modeled, class design may be complex. An important principle in class design is to ensure that an object is never put into an *inconsistent state* by responding to a message.

statement

The basic building block of a Java method. There are many different types of statement in Java, for instance, the *assignment statement*, *if statement*, *return statement* and *while loop*.

statement terminator

The semicolon (`;`) is used to indicate the end of a statement.

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.

string

An instance of the `String` class. Strings consist of zero or more *Unicode* characters, and they are *immutable*, once created. A literal string is written between a pair of string delimiters (`"`), as in

```
"hello, world"
```

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*. See *sub type*

super class

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

switch statement

A selection statement in which the value of an *arithmetic expression* {expression!arithmetic} is compared for a match against different *case labels*. If no match is found, the optional *default label* is selected For instance

```
switch(choice){
     case 'q':
         quit();
         break;
     case 'h':
         help();
         break;
     ...
```

```
            default:
                System.out.println("Unknown command: "+choice);
                break;
        }
```

syntax error

An error detected by the *compiler* during its *parsing* of a program. Syntax errors usually result from mis-ordering symbols within expressions and statements. Missing curly brackets and semicolons are common examples of syntax errors.

T

ternary operator

See *conditional operator*

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);
```

U

unary operator

An *operator* taking a single operand. Java's unary operators are `-`, `+`, `!`, `!`, `++` and `--`.

---

uninitialized variable

A local variable that been declared, but has had no value assigned to it. The compiler will warn of variables which are used before being initialized.

V

variable declaration

The association of a variable with a particular type. It is important to make a distinction between the declaration of variables of primitive types and those of class types. A variable of primitive type acts as a container for a single value of its declared type. Declaration of a variable of a class type does not automatically cause an object of that type to be constructed and, by default, the variable will contain the value `null`. A variable of a class type acts as a holder for a reference to an object that is compatible with the variable's class type. Java's rules of *polymorphism* allow a variable of a class type to hold a reference to any object of its declared type or any of its sub types. A variable with a declared type of `Object`, therefore, may hold a reference to an object of any class, therefore.

W

while loop

One of Java's three *control structures* used for looping. The other two are the *do loop* and *for loop*. A while loop consists of a *boolean expression* and a loop body. The condition is tested before the loop body is entered for the first time and re-tested each time the end of the body is completed. The loop terminates when the condition gives the value `false`. The statements in the loop body might be executed zero or more times.

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.
