[**abstract**](http://www.codejava.net/java-core/the-java-language/abstract-keyword)

The **abstract** keyword is used to declare a class or a method as abstract. An abstract class can have abstract methods which have to be implemented by its sub classes. An abstract method does not have concrete implementation or body, and must ends with a semicolon. The following example declares a class Car as abstract:

|  |  |
| --- | --- |
| 1  2  3 | public abstract class Car {      public abstract void drive();  } |

The method drive() is also declared as abstract.

**Rules for abstract class and abstract method:**

An abstract class:

* + cannot be instantiated by **new** keyword. The purpose of an abstract class is to be inherited by derived classes.
  + can have both abstract and non-abstract methods.

 An abstract method:

* + does not have body and must end with a semicolon.
  + must be implemented by concrete sub classes.
  + makes the enclosing class must be declared as abstract also

**Examples**

A class that extends the abstract Car class and implements its abstract method:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class PoliceCar extends Car {      public void drive() {          // drive faster than normal car      }  } |

An abstract class has both abstract and non-abstract methods:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | abstract class Airplane {      public abstract void takeOff();      public void landing() {          // landing smoothly      }  } |

# [assert](http://www.codejava.net/java-core/the-java-language/java-keyword-assert)

The **assert** keyword is used in assert statement which is a feature of the Java programming language since Java 1.4. Assertion enables developers to test assumptions in their programs as a way to defect and fix bugs.

# Syntax of assert statement

 Syntax of an assert statement is as follow (short version):

**assert***expression1*;

or (full version):

**assert** *expression1* : *expression2*;

Where:

* *expression1* must be a boolean expression.
* *expression2* must return a value (must not return void).

The assert statement is working as follows:

* + If assertion is enabled, then the assert statement will be evaluated. Otherwise, it does not get executed.
  + If *expression1* is evaluated to false, an AssertionError error is thrown which causes the program stops immediately. And depending on existence of *expression2*:
    - If *expression2* does not exist, then the AssertionError is thrown with no detail error message.
    - If *expression2* does exist, then a String representation of *expression2*’s return value is used as detail error message.
  + If *expression1* is evaluate to true, then the program continues normally.

# Enable assertion

By default, assertion is disabled at runtime. To enable assertion, specify the switch –enableassertions or -ea at command line of java program. For example, to enable assertion for the program called CarManager:

**java –enableassertions CarManager**

or this for short:

**java –ea CarManager**

Assertion can be enabled or disable specifically for named classes or packages. For more information on how to enable and disable assertion, go to: [http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html#enable-disable](http://www.codejava.net/java-core/the-java-language/java-keyword-assert#enable-disable)

# Assertion examples

 The following simple program illustrates the short version of assert statement:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class AssertionExample {      public static void main(String[] args) {          // get a number in the first argument          int number = Integer.parseInt(args[0]);            assert number <= 10; // stops if number > 10            System.out.println("Pass");        }  } |

 When running the program above with this command:

**java -ea AssertionExample 15**

A java.lang.AssertionError error will be thrown:

Exception in thread "main" java.lang.AssertionError

       at AssertionExample.main(AssertionExample.java:6)

But the program will continue and print out “Pass” if we pass a number less than 10, in this command:

**java -ea AssertionExample 8**

And the following example is using the full version of assert statement:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class AssertionExample2 {      public static void main(String[] args) {            int argCount = args.length;            assert argCount == 5 : "The number of arguments must be 5";            System.out.println("OK");        }  } |

 When running the program above with this command:

**java -ea AssertionExample2 1 2 3 4**

it will throw this error:

Exception in thread "main" java.lang.AssertionError: The number of arguments must be 5

       at AssertionExample2.main(AssertionExample2.java:6)

Generally, assertion is enabled during development time to defect and fix bugs, and is disabled at deployment or production to increase performance.

Top of Form

Bottom of Form

# [boolean](http://www.codejava.net/java-core/the-java-language/boolean-keyword)

The **boolean** keyword is used to declare a variable as a boolean type, which represents only either **true** or **false**. For example:

boolean isAlive;

The boolean keyword can be used to declared return type of a method as well:

public boolean hasChildren() { return true; }

# [break](http://www.codejava.net/java-core/the-java-language/break-keyword)

The **break** keyword stops execution of [for loop](http://www.codejava.net/java-core/the-java-language/for-keyword), [while loop](http://www.codejava.net/java-core/the-java-language/do-while-construct) and [switch-case](http://www.codejava.net/java-core/the-java-language/switch-case-construct) construct. The execution goes to next statement after the construct is broken.

The following example shows a **break** statement inside a for loop:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | for (int count = 0; count < 100; count++) {      System.out.println("count = " + count);      if (count > 70) {          break;      }  } |

The above for loop prints values of count from 0 to 70.

The following example shows **break** statement is used in a while loop:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int number = 0;  while (true) {      number++;      System.out.println("number = " + number);      if (number > 1000) {          break;      }  } |

The above while loop will stop when the value of number is greater than 1000.

The following example shows **break** statements are used in every case of the switch statement:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | int monthNumber = 2;  String monthName = "";  switch(monthNumber) {      case 1:          monthName = "January";          break;      case 2:          monthName = "February";          break;      case 3:          monthName = "March";          break;      case 4:          monthName = "April";          break;  }  System.out.println("The month is " + monthName); |

The above code will produce the output: The month is February. The **break** statements prevent the execution from *falling through* from one case to another.

# [byte](http://www.codejava.net/java-core/the-java-language/byte-keyword)

The **byte** keyword is used to declared a variable as a numeric type. A **byte** value can hold an 8-bit integer number which ranges from -128 to 127. For example:

|  |  |
| --- | --- |
| 1 | byte month = 6; |

The **byte** keyword can be used to declared return type of a method as well:

|  |  |
| --- | --- |
| 1  2  3 | public byte getMonth() {      return 12;  } |

# [try-catch-finally construct](http://www.codejava.net/java-core/the-java-language/try-catch-finally-construct)

To handle error/exceptions which may occur during execution of a particular code, Java introduces **try-catch-finally**construct. The structure of this construct is as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | try {      // code that may throws errors/excpetions  } catch (exception object is being thrown) {      // code to handle exception  } finally {      // code always executed, regardless of exception thrown or not,      // i.e clean up resources  } |

For example, the following code catch an exception may be thrown when parsing a number from an input string:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | String input = "";  // capture input from user...    int number;  try {      number = Integer.parseInt(input);  } catch (NumberFormatException ex) {      number = -1;    // assigns  a default value  } |

**Rules**

The catch block can be optional. The following example show only a **try-finally** construct:

|  |  |
| --- | --- |
| 1  2  3  4  5 | try {      number = Integer.parseInt(input);  } finally {      number += 100;  } |

The finally block is also optional, as shown in the first example. But remember that code in finally block is always executed, regardless of whether the try block throwing exception or not.

# [Java keyword: char](http://www.codejava.net/java-core/the-java-language/char-keyword)

The **char** keyword is used to declared a variable as a character type. A **char** variable represents a single character. For example:

|  |  |
| --- | --- |
| 1 | char letterA = 'A'; |

The **char** keyword can be used to declared return type of a method as well:

|  |  |
| --- | --- |
| 1  2  3 | public char getALetter() {      return 'A';  } |

# [class](http://www.codejava.net/java-core/the-java-language/class-keyword)

The **class** keyword is used to declare a class, for example:

|  |  |
| --- | --- |
| 1  2  3  4 | class Person {      // properties...      // methods...  } |

**Const**

# [continue](http://www.codejava.net/java-core/the-java-language/continue-keyword)

The **continue** keyword is used to stop execution of a current iteration in a [for loop](http://www.codejava.net/java-core/the-java-language/for-keyword) or a [while loop](http://www.codejava.net/java-core/the-java-language/do-while-construct), then advance to the next iteration. The statements after the continue keyword won’t be executed. The syntax is as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | for/while (expressions) {      // statemenst 1...      if (condition) {          continue;      }      // statements 2...  } |

The following example uses **continue**keyword to skip the even value of the variable i, thus the for loop will print out all the odd numbers from 1 to 100:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | for (int i = 1; i < 100; i++) {      if (i % 2 == 0) {          continue;      }      System.out.println(i);  } |

Similarly, the following **while**loop will produce a list of even numbers from 1 to 100:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | int count = 1;  while (count <= 100) {      if (count % 2 != 0) {          count++;          continue;      }      System.out.println(count);      count++;  } |

# [do-while construct](http://www.codejava.net/java-core/the-java-language/do-while-construct)

The **do-while** construct is a loop structure which repeatedly executes one or some statements until a condition becomes false. In other words, it repeats the statements while the condition is still true. There are two forms:

The first form (there is only **while** keyword):

|  |  |
| --- | --- |
| 1  2  3 | while (condition is true) {      // statements  } |

The second form (there are both **do** and **while** keywords):

|  |  |
| --- | --- |
| 1  2  3 | do {      // statements  } while (condition is true); |

# [double](http://www.codejava.net/java-core/the-java-language/double-keyword)

The **double** keyword is used to declared a variable as a numeric type. A **double** value can hold a 64-bit floating-point number. For example:

|  |  |
| --- | --- |
| 1 | double balanceAmount; |

The **double** keyword can be used to declared return type of a method as well:

|  |  |
| --- | --- |
| 1  2  3 | public double getBalance() {      return balanceAmount;  } |

# [if-else construct](http://www.codejava.net/java-core/the-java-language/if-else-construct)

The **if-else** construct is used to check if a condition is met then do something, otherwise do something else. The structure looks like follows:

|  |  |
| --- | --- |
| 1  2  3  4  5 | if (condition) {      // do something if condition is met  } else {      // do something else if condition is not met  } |

where condition can be a variable, a method call or an expression that must be evaluated to true. The code inside if block can be any Java statements. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | boolean passed = false;  // try something...  if (passed) {      System.out.println("Congratulations!");  } else {      System.out.println("Try again next time!");  } |

The construct can be extended for multiple if-else pairs:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | if (condition1) {      // do something if condition1 is met  } else if (condition2) {      // do something if condition2 is met  } else if (condition3) {      // do something if condition3 is met  } else {      // do something else if none condition is met  } |

For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int score = calculateScore();    if (score > 90) {      System.out.println("Excellent");  } else if (score > 70) {      System.out.println("Very good");  } else if (score > 50) {      System.out.println("Above average");  } else {      System.out.println("Very bad");  } |

If there is only one statement after the **if** or **else**, the opening and closing curly braces can be removed. For example:

|  |  |
| --- | --- |
| 1  2  3  4 | if (score < 10)      score = score + 10;  else      score = score - 10; |

# [enum](http://www.codejava.net/java-core/the-java-language/enum-keyword)

The **enum** keyword is used to declare a new enumeration type. This keyword has been introduced since Java 5. The declaration syntax for an enum type is as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | enum <name> {      <enum\_constant\_1>,      <enum\_constant\_2>,      ...      <enum\_constant\_n>      <enum\_constructor>      // other variables & methods as usual  } |

An enum constant can be declared as follows:

|  |  |
| --- | --- |
| 1 | <constant\_name>[(arguments)] [{class body}] |

# Rules for enum type

* + An enum constant specifies an instance of the enum type.
  + An enum constant can be optionally followed by a list of arguments and a class body. The class body is an anonymous class which conforms to rules of anonymous classes and:
    - It cannot have any constructor.
    - It cannot have any abstract methods.
    - Instance methods declared in the class body are only accessible if they override accessible methods declared in the enclosing enum type.
  + An enum type cannot be declared abstract or final.
  + The Enum<E> is the direct superclass of an enum type.
  + An enum type can be only declared inside class level, same as class level or in a separate source file. It cannot be declared inside a method or an inner class.
  + An enum type can have constructors, methods and variables just like a regular Java class.

# Examples for enum type

A simplest enum type declared inside a class:

|  |  |
| --- | --- |
| 1  2  3 | class Foo {      enum DayOfWeek {MON, TUE, WED, THU, FRI, SAT, SUN};  } |

A simple enum type declared in a separate Java file:

|  |  |
| --- | --- |
| 1  2  3 | public enum ErrorCode {      LOW, HIGH, SEVERE  } |

An enum type which contains constructor, method and variable:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | enum Day {      MON(1), TUE(2), WED(3), THU(4), FRI(5), SAT(6), SUN(7);        Day(int dayNumber) {          this.dayNumber = dayNumber;      }        private int dayNumber;        public int getDayNumber() {          return this.dayNumber;      }  } |

An enum type with class body for each enum constant:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [extends](http://www.codejava.net/java-core/the-java-language/extends-keyword) When declaring a class as a subclass of another class, the **extends** keyword is used. Here is an example:   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12 | class A {        int value;        void doSomething() {        }  }    class B extends A {    } |   In the above example, class B is a subclass of class A. In other words, class A is a super class of class B. As a sub class, class B inherits (extends) variables and methods declared from class A.  The extends keyword is also used to make an interface extends another interface, for example:   |  |  | | --- | --- | | 1  2  3 | interface Animal { }    interface Reptile extends Animal { } | | enum Priority {      LOW {          int getPriorityNumber() {              return 0;          }      },        NORMAL {          int getPriorityNumber() {              return 1;          }      },        HIGH {          int getPriorityNumber() {              return 2;          }      },        SEVERE {          int getPriorityNumber() {              return 3;          }      };        abstract int getPriorityNumber();  } |

# [final](http://www.codejava.net/java-core/the-java-language/final-keyword)

The **final** keyword can be applied to declaration of classes, methods and variables.

* **Final class**: if a class is marked as final, it cannot be subclassed/inherited by another class. For example:

|  |  |
| --- | --- |
| 1 | final class A { } |

* then the following code will not compile:

|  |  |
| --- | --- |
| 1 | class B extends A {} // compile error |

* **Final method**: when a method is final, that means it cannot be overridden, neither by methods in the same class or in sub class. For example:

|  |  |
| --- | --- |
| 1  2  3 | class C {      final void foo() { }  } |

* the subclass D attempts to override the method foo(), but fail because foo() is marked as final:

|  |  |
| --- | --- |
| 1  2  3 | class D extends C {      void foo() { } // compile error  } |

* **Final variable**: if a variable is marked as final, its reference cannot be changed to refer to another object, once initialized. For example:

|  |  |
| --- | --- |
| 1 | final String message = "HELLO"; |

* Once the variable message is initialized and marked as final, the following code attempts to assign another value to it, will fails:

|  |  |
| --- | --- |
| 1 | message = "BONJOUR";    // compile error |

**Note**: a class cannot be both abstract and final.

# [float](http://www.codejava.net/java-core/the-java-language/float-keyword)

The **float** keyword is used to declared a variable as a numeric type. A **float** value can hold a 32-bit floating-point number. For example:

|  |  |
| --- | --- |
| 1 | float balanceAmount = 909.999; |

The **float** keyword can be used to declared return type of a method as well:

|  |  |
| --- | --- |
| 1  2  3 | public float getBalance() {      return balanceAmount;  } |

# [for](http://www.codejava.net/java-core/the-java-language/for-keyword)

The **for** keyword is used to iterate over a range of values, items of an array or a collection. The code that uses **for** keyword is called as for loop.

In Java, there are two types of for loop:classic for loop and enhanced for loop (for-each).

# ****Classic for loop****

Syntax of the classic for loop is as follows:

for (initialization; terminate condition; incrementation) {

    // statements

}

where:

* + initialization: contains code that initializes the loop, usually by declaring a counter variable. This code is executed only once.
  + terminate condition: contains code that checks for a condition to terminate the loop. This code is executed every iteration, if the codition is met, the loop is terminated.
  + incrementation: contains code that increments value of the counter variable. This code is executed every iteration.

The following code example iterates over a range of integer numbers from 0 to 9 and output the current number to the standard output:

|  |  |
| --- | --- |
| 1  2  3 | for (int i = 0; i < 10; i++) {      System.out.println("Number: " + i);  } |

**goto**

# [implements](http://www.codejava.net/java-core/the-java-language/implements-keyword)

The **implements** keyword is used to make a class adheres to contract defined by an interface. The implemented class must provide concrete implementation for the methods defined by the interface. If not, the class must be abstract.

The following example illustrates a class implements an interface and provides detailed implementation for the interface's methods:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | interface Vehicle {      void start();      void stop();  }    class Car implements Vehicle {      void start() {          // starts the engine      }        void stop() {          // stop the engine      }  } |

Unlike [extends](http://www.codejava.net/index.php/java-core/87-the-java-language/104-extends-keyword) keyword, a class can implement multiple interfaces. The interface names are separated by commas. For example:

|  |  |
| --- | --- |
| 1  2  3 | interface Peson { }  interface Employee {}  class Director implements Person, Employee { } |

If the implemented class does not provide implementation for the interface's methods, the class must be abstract. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | interface Animal {      void eat();  }    abstract class Reptile implements Animal {      abstract void crawl();  } |

In that case, the first non-abstract class must implement the method. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Crocodile extends Reptile {      void eat() {          // eats      }        void crawl() {          // crawls      }  } |

# [import](http://www.codejava.net/java-core/the-java-language/import-keyword)

The **import** keyword is used to make classes and interfaces available and accessible to the current source code, without specifying fully qualified package names. For example:

|  |  |
| --- | --- |
| 1  2  3 | import java.awt.\*;  import java.util.List;  import java.io.File; |

The first import statement makes all classes/interfaces under package java.awt accesible to the current program. It uses the **\*** wildcard character to specify 'everything' under the package.

Likewise, the second and third import statements make the interface List and the class File available to the current source code. It uses exact names instead of wildcard.

The import statements must be placed on top of the source file, only after the package statement.

# [instanceof](http://www.codejava.net/java-core/the-java-language/instanceof-keyword)

The **instanceof** keyword is used to check whether an object is an instance of a particular class or interface. For example:

|  |  |
| --- | --- |
| 1  2  3  4 | Object msg = new String("Hello");  if (msg instanceof String) {      System.out.println("A String");  } |

# [interface](http://www.codejava.net/java-core/the-java-language/interface-keyword)

The **interface** keyword is used to declare an interface:

|  |  |
| --- | --- |
| 1  2  3  4 | interface Vehicle {      void start();      void stop();  } |

Interface is a programming construct that defines a contract which must be adhered by implementing classes. The methods defined in an interface must have no body and end with a semicolon. The implementing classes will provide concrete implementation for the methods.

# [native](http://www.codejava.net/java-core/the-java-language/native-keyword)

The **native** keyword is used to declare a method which is implemented in platform-dependent code such as C or C++. When a method is marked as native, it cannot have a body and must ends with a semicolon instead. The [Java Native Interface (JNI)](http://docs.oracle.com/javase/7/docs/technotes/guides/jni/)specification governs rules and guidelines for implementing native methods, such as data type conversion between Java and the native application.

The following example shows a class with a method declared as native:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class NativeExample {        public native void fastCopyFile(String sourceFile, String destFile);    } |

# [strictfp](http://www.codejava.net/java-core/the-java-language/java-keyword-strictfp)

The **strictfp** keyword is used to force the precision of floating point calculations (float or double) in Java conform to [IEEE’s 754 standard](http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=2355), explicitly. Without using **strictfp** keyword, the floating point precision depends on target platform’s hardware, i.e. CPU’s floating point processing capability. In other words, using **strictfp** ensures result of floating point computations is always same on all platforms.

The **strictfp** keyword can be applied for classes, interfaces and methods.

**Rules**

* + **strictfp** cannot be applied for constructors.
  + If an interface or class is declared with **strictfp**, then all methods and nested types within that interface or class are implicitly **strictfp**.
  + **strictfp** cannot be applied for interface methods.

**Examples**

The following class is declared with **strictfp**, hence all the floating point computations within that class conform to IEEE’s 754 standard:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | strictfp class StrictFPClass {      double num1 = 10e+102;      double num2 = 6e+08;      double calculate() {          return num1 + num2;      }  } |

The following interface is declared with **strictfp**, but its methods cannot:

|  |  |
| --- | --- |
| 1  2  3  4 | strictfp interface StrictFPInterface {      double calculate();      strictfp double compute();    // compile error  } |

The following method is declared with **strictfp**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class StrictFPMethod {      strictfp double computeTotal(double x, double y) {          return x + y;      }  } |

# [super](http://www.codejava.net/java-core/the-java-language/super-keyword)

The **super** keyword is used to access variables and methods of a super class from a sub class. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class Super {      protected int number;        protected showNumber() {          System.out.println("number = " + number);      }  }    public class Sub extends Super {      void bar() {          super.number = 10;          super.showNumber();      }  } |

In the above example, the class Sub accesses the variable number and calls the method showNumber() for its super class Super.

# [switch-case construct](http://www.codejava.net/java-core/the-java-language/switch-case-construct)

The **switch-case** construct is a flow control structure that tests value of a variable against a list of values. Syntax of this structure is as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | switch (expression) {      case constant\_1:          // statement 1          break;      case constant\_2:          // statement 2          break;      case constant\_3:          // statement 3          break;      //...      case constant\_n:          // statement n          break;      default:          // if all the cases do not match  } |

**Rules**

* + The expression is a variable or an expression which must be evaluated to one of the following types:
    - Primitive numbers: byte, short, char and int.
    - Primitive wrappers: Byte, Short, Character and Integer.
    - Enumerated types (enum type).
    - [String objects](http://www.codejava.net/java-core/the-java-language/using-strings-in-switch-case-statement-java-7) (since Java 1.7)
  + The constant\_1, constant\_2, constant\_3, …, constant\_n must be a constant or literals of the allowed types.
  + Each case is tested from top to bottom, until a case is matched and a break statement is found.
  + If a case matches the *expression*, the statements block after the case clause are executed, until a break statement is reached.
  + It is not required that each case must have a corresponding break statement. If a matching case block does not have a break statement, the execution will *fall through* the next case block, until a first break statement is reached or end of switch statement is encountered.
  + The statements block after default will be executed if there is no matching case found.

**Examples**

* + The following example shows a switch statement that tests for an integer variable. It products the output: The number is Three

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int number = 3;  String text = "";  switch (number) {      case 1:          text = "One";          break;      case 2:          text = "Two";          break;      case 3:          text = "Three";          break;      default:          text = "Other number";  }  System.out.println("The number is: " + text); |

* + The following example shows a switch statement that tests for a String variable, without a default block. It will output: The distance from earth to Jupiter is: 4

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | String planet = "Jupiter";  long distanceFromEarth = 0;  switch (planet) {      case "Mars":          distanceFromEarth = 3;          break;      case "Saturn":          distanceFromEarth = 5;          break;      case "Jupiter":          distanceFromEarth = 4;          break;      case "Venus":          distanceFromEarth = 1;          break;  }  System.out.println("The distance from earth to " + planet + " is: " + distanceFromEarth); |

# [synchronized](http://www.codejava.net/java-core/the-java-language/synchronized-keyword)

The **synchronized** keyword is used for code blocks and methods where thread-safe matters and for multi-threaded (concurrent) programming. A synchronized method or a synchronized statement can be executed by only one thread at a time.

The syntax for a **synchronized** **method** is as follows:

|  |  |
| --- | --- |
| 1  2  3 | <method modifier> synchronized <method signature> {      // synchronized code block  } |

The syntax for a **synchronized** **statement** is as follows:

|  |  |
| --- | --- |
| 1  2  3 | synchronized (expression) {      // synchronized code block  } |

**Rules**

-          The expression must be evaluated to a reference type, i.e an object reference.

-          The current executing thread will try to acquire a lock before executes the synchronized code block:

* + - * + For synchronized statement: the monitor associates with the object returns by the expression.
        + For synchronized method:

If the method is static, the lock associates with the Class object of the class in which the method is declared.

If the method is non-static, the lock associates with this – the object for which the method is invoked.

-          If the lock is not acquired by any thread, the current executing thread will own the lock and execute the synchronized code block.

-          While the current executing thread owns the lock, no other threads can acquire that lock.

-          When the synchronized code block completes, the current executing thread releases the lock.

-          There is no something called “synchronized constructor”.

**Examples**

The following code example illustrates the synchronized keyword is applied for a static method:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class Counter {      private static int count;      static synchronized void increase() {          count++;      }  } |

The following code example shows instance methods are synchronized:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class BankAccount {      private double balance;      synchronized void withdraw(double amount) {          this.balance -= amount;      }      synchronized void deposit(double amount) {          this.balance += amount;      }  } |

The following code example shows a synchronized statement is applied for a code block, not a method:

|  |  |
| --- | --- |
| 1  2  3  4 | Object lock = new Object();  synchronized (lock) {      System.out.println("Synchronized statement");  } |

Top of Form

Bottom of Form

# [this](http://www.codejava.net/java-core/the-java-language/this-keyword)

The **this** keyword is used to refer to the current instance of a class. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Manager {      Employees[] employees;        void manageEmployees() {          int totalEmp = this.employees.length;          System.out.println("Total employees: " + totalEmp);          this.report();      }        void report() { }  } |

In the above example, the **this** keyword is used in two places:

* **this**.employees.length: accesses a variable of the current instance of class Manager.
* **this**.report(): invokes a method of the current instance of the class Manager.

The **this** keyword is optional, that means if the above example will behave the same if it does not use **this** keyword. However, using **this** keyword may make the code more readable or understandable.

# [throw and throws](http://www.codejava.net/java-core/the-java-language/throw-and-throws-keywords)

The **throw** keyword is used to throw an exception from within a method. When a **throw** statement is encountered and executed, execution of the current method is stopped and returned to the caller.

Whereas the **throws** keyword is used to declare that a method may throw one or some exceptions. The caller has to catch the exceptions (catching is optional if the exceptions are of type unchecked exceptions).

These two keywords are usually used together as depicted the following form:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | void aMethod() throws Exception1, Exception2 {        // statements...      if (an exception occurs) {          throw new Exception1();      }        // statements...      if (another exception occurs) {          throw new Exception2();      }  } |

**Rules**

* + *Exception1, Exception2, …*: the exception class must be a direct subclass of Throwable class one of its subclasses.
  + The **throws** keyword can be followed by one more exception class, separated by commas.
  + The **throw** keyword must be followed by an instance of Throwable class or one of its subclasses.
  + When using the **throw** keyword to throw a checked exception from within a method, the method must either:
    - * Declares the **throws** clause followed by the exceptions thrown by the **throw** statements, or:
      * Catches the exceptions thrown by the **throw** statements.
  + When a method contains statements which may throw exceptions (not using **throw** statements explicitly), it also has to either catch or declare to throw the exceptions.
  + If the **throw** statements throw unchecked exceptions, the method is not required to declare those unchecked exceptions in its **throws** clause.
  + A concrete method can declare **throws** clause if only if its body throws checked exceptions. Otherwise a compile error occurs.
  + An interface’s method can declare **throws** clause freely.
  + The **throws** clause can declare exceptions which are super types of the exception thrown by the **throw** statements, but not sub types.

**Code Examples**

An interface declares a method that throws an exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | interface AutoMobile {        void startEngine() throws EngineStartException;        void go();  } |

Where EngineStartException is a subclass of Exception class whose super type is Throwable:

|  |  |
| --- | --- |
| 1  2 | class EngineStartException extends Exception {  } |

Bottom of Form

# [transient](http://www.codejava.net/java-core/the-java-language/transient-keyword)

The **transient** keyword is used to mark a member variable (field) not a part of the persistent state of an object, i.e. the field will not be persisted when its parent object is serialized.

The following example shows a class which has a transient variable:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Rectangle {      int width;      int height;      transient long area;  } |

When an object of Rectangle class is being serialized, the fields width and height will be persisted, but the field area.

The **transient** keyword can be only applied for member variable, not for class, method and local variable.

# [volatile](http://www.codejava.net/java-core/the-java-language/volatile-keyword)

The **volatile** keyword can be applied for only member variables (fields). When a volatile variable is accessed concurrently by threads, its value is updated consistently among threads. In some cases, using volatile can be an alternative to using [synchronized](http://www.codejava.net/java-core/the-java-language/synchronized-keyword) code.

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5 | class VolatileExample {        volatile int x;    } |

**Rules**

* The **volatile** keyword cannot be applied for class, method and local variable.
* A *final* variable cannot be declared volatile.