## Keywords

## Identifiers

Identifiers are used to name elements of Java programs, such as classes, interfaces, variables, methods, constructors, and arguments. A valid name begins with a Unicode letter, underscore character (\_), or dollar sign ($). A Unicode letter is an English letter (a-z) or (A-Z) or a letter in a non-English language (There are too many to list). The second and remaining letters of an identifier may be a Unicode letter, underscore, dollar sign, or digit (0-9). The dollar sign character is used by compiler-internal identifiers and should be avoided.   
Identifiers are case sensitive. The identifiers myID and MyID are considered to be separate identifiers. An identifier may not use the literal values, true, false, null, or any of the keywords that are reserved by Java.

47 Java Keywords

The Java language reserves a list of 47 keywords.   
[Keywords may not be used as identifiers](http://www.j2eeonline.com/java2-programmers-certification/module2/keywords-literal-values.jsp).

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

## Data types

A primitive is the most basic form of data in a Java program, hence the name

Primitive

int (integer),

float (floating point), boolean (boolean), and char (character).

The remaining

four are byte, short, long, and double.

Calculations performed with primitives are much faster than calculations performed

with similar objects.

Almost everything you work with in Java is an object.

Primitives are one of the few exceptions to this rule.



## Assignment statements

The general usage of the assignment statement:

variable = value;

## Conditional

if (expression)

statementA;

else

statementB;

The general usage of the if-then statement:

if (expressionA)

statementA;

else if (expressionB)

statementB;

The general usage of the if-then-else statement:

if (expressionA)

statementA;

else if (expressionB)

statementB;

else if (expressionC)

statementC;

…

else

statementZZ;

The general usage of the switch statement:

switch (expression) {

case valueA:

// Sequences of statements

break;

case valueB:

// Sequences of statements

break;

default:

// Sequences of statements

…

}

The expression of the switch statement must evaluate to byte, short, int,

or char. Wrapper classes of type Byte, Short, Int, and Character are also

allowed

## Iteration

The general usage of the for statement:

for ( initialization; expression; iteration) {

// Sequence of statements

}

The general usage of the for statement:

for (type variable : collection) statement-sequence

int hookSizes[] = { 1, 1, 1, 2, 2, 4, 5, 5, 5, 6, 7, 8, 8, 9 };

for (int hook: hookSizes) System.out.print(hook + " ");

$ 1 1 1 2 2 4 5 5 5 6 7 8 8 9

The general usage of the while statement:

while (expression) {

// Sequences of statements

}

do {

// Sequence of statements

} while (expression)

Jump statements-

Break

Continue

Return

## Performing Code Refactoring??



## Assignment operators

## Arithmetic operators

## Relational operators

## Logical operators

## Classes and objects

A class is like

a blueprint to tell the Java Virtual Machine how to create an object at runtime.

The new operator tells the Java Virtual Machine to create a new instance of this

class, the result of which is an object.

The new operator will always return a new and

independent instance of the class.

When an object is declared, the Java Virtual Machine makes a reference

in memory to the location of the object. Objects also have the ability to change

in size as the objects they store grow or shrink. An object is declared in the same

manner as a primitive, but cannot be used until it has been initialized with the new

operator or set equal to an existing object.

Encapsulation and abstraction-

Encapsulation is a mechanis that binds together code and the data it manipulates and keeps both safe from outside interference. It provides an explicit boundary between object;s abstract interfaces (functions) and its internal implementations (data in general).

Classes are abstract data types.

## Access modifiers

The four access levels are:

Visible to the package. the default. No modifiers are needed.

Visible to the class only (private).

Visible to the world (public).

Visible to the package and all subclasses (protected).

A variable or method declared without any access control modifier is available to any other class in the same package. The default modifier cannot be used for methods, fields in an interface.

Private Access Modifier - private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

## Protected Access Modifier - protected:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Class members-

Data Members- static and non static

Member functions- static, non static, constructors.

## Constructors- Default, parameterized, copy.

Java provides a default constructor which takes no arguments and performs no special actions or initializations, when no explicit constructors are provided.

The only action taken by the implicit default constructor is to call the superclass constructor using the super() call.

public class Cube1 {

int length;

int breadth;

int height;

public int getVolume() {

return (length \* breadth \* height);

}

Cube1() {

length = 10;

breadth = 10;

height = 10;

}

Cube1(int l, int b, int h) {

length = l;

breadth = b;

height = h;

}

public static void main(String[] args) {

Cube1 cubeObj1, cubeObj2;

cubeObj1 = new Cube1();

cubeObj2 = new Cube1(10, 20, 30);

System.out.println("Volume of Cube1 is : " + cubeObj1.getVolume());

System.out.println("Volume of Cube1 is : " + cubeObj2.getVolume());

}

}

|  |
| --- |
| public class Cube2 {  int length;  int breadth;  int height;  public int getVolume() {  return (length \* breadth \* height);  }  Cube2() {  this(10, 10);  System.out.println("Finished with Default Constructor");  }  Cube2(int l, int b) {  this(l, b, 10);  System.out.println("Finished with Parameterized Constructor having 2 params");  }  Cube2(int l, int b, int h) {  length = l;  breadth = b;  height = h;  System.out.println("Finished with Parameterized Constructor having 3 params");  }  public static void main(String[] args) {  Cube2 cubeObj1, cubeObj2;  cubeObj1 = new Cube2();  cubeObj2 = new Cube2(10, 20, 30);  System.out.println("Volume of Cube1 is : " + cubeObj1.getVolume());  System.out.println("Volume of Cube2 is : " + cubeObj2.getVolume());  }  }  public class Cube2 {  int length;  int breadth;  int height;  public int getVolume() {  return (length \* breadth \* height);  }  Cube2() {  this(10, 10);  System.out.println("Finished with Default Constructor");  }  Cube2(int l, int b) {  this(l, b, 10);  System.out.println("Finished with Parameterized Constructor having 2 params");  }  Cube2(int l, int b, int h) {  length = l;  breadth = b;  height = h;  System.out.println("Finished with Parameterized Constructor having 3 params");  }  public static void main(String[] args) {  Cube2 cubeObj1, cubeObj2;  cubeObj1 = new Cube2();  cubeObj2 = new Cube2(10, 20, 30);  System.out.println("Volume of Cube1 is : " + cubeObj1.getVolume());  System.out.println("Volume of Cube2 is : " + cubeObj2.getVolume());  }  } |

Output

Finished with Parameterized Constructor having 3 params  
Finished with Parameterized Constructor having 2 params  
Finished with Default Constructor  
Finished with Parameterized Constructor having 3 params  
Volume of Cube1 is : 1000  
Volume of Cube2 is : 6000

## Constructor Chaining

Every constructor calls its superclass constructor. An implied super() is therefore included in each constructor which does not include either the this() function or an explicit super() call as its first statement. The super() statement invokes a constructor of the super class.  
The implicit super() can be replaced by an explicit super(). The super statement must be the first statement of the constructor.  
The explicit super allows parameter values to be passed to the constructor of its superclass and must have matching parameter types A super() call in the constructor of a subclass will result in the call of the relevant constructor from the superclass, based on the signature of the call. This is called constructor chaining.

Below is an example of a class demonstrating constructor chaining using super() method.

|  |
| --- |
| class Cube {  int length;  int breadth;  int height;  public int getVolume() {  return (length \* breadth \* height);  }  Cube() {  this(10, 10);  System.out.println("Finished with Default Constructor of Cube");  }  Cube(int l, int b) {  this(l, b, 10);  System.out.println("Finished with Parameterized Constructor having  2 params of Cube");  }  Cube(int l, int b, int h) {  length = l;  breadth = b;  height = h;  System.out.println("Finished with Parameterized Constructor having  3 params of Cube");  }  }  public class SpecialCube extends Cube {  int weight;  SpecialCube() {  super();  weight = 10;  }  SpecialCube(int l, int b) {  this(l, b, 10);  System.out.println("Finished with Parameterized Constructor having  2 params of SpecialCube");  }  SpecialCube(int l, int b, int h) {  super(l, b, h);  weight = 20;  System.out.println("Finished with Parameterized Constructor having  3 params of SpecialCube");  }  public static void main(String[] args) {  SpecialCube specialObj1 = new SpecialCube();  SpecialCube specialObj2 = new SpecialCube(10, 20);  System.out.println("Volume of SpecialCube1 is : "  + specialObj1.getVolume());  System.out.println("Weight of SpecialCube1 is : "  + specialObj1.weight);  System.out.println("Volume of SpecialCube2 is : "  + specialObj2.getVolume());  System.out.println("Weight of SpecialCube2 is : "  + specialObj2.weight);  }  } |

Output

Finished with Parameterized Constructor having 3 params of SpecialCube  
Finished with Parameterized Constructor having 2 params of SpecialCube  
Volume of SpecialCube1 is : 1000  
Weight of SpecialCube1 is : 10  
Volume of SpecialCube2 is : 2000  
Weight of SpecialCube2 is : 20

The super() construct as with this() construct: if used, must occur as the first statement in a constructor, and it can only be used in a constructor declaration. This implies that this() and super() calls cannot both occur in the same constructor. Just as the this() construct leads to chaining of constructors in the same class, the super() construct leads to chaining of subclass constructors to superclass constructors.  
if a constructor has neither a this() nor a super() construct as its first statement, then a super() call to the default constructor in the superclass is inserted.

Enumerations are a special data type in Java that allows for a variable to be set to

predefined constants. The variable must equal one of the values that have been

predefined for it. An enumeration is useful when there is a limited set of options that

a variable can equal and it is restricted to these known values. For example, a deck

of playing cards will always have four suits: clubs, diamonds, hearts, and spades. If a

developer wanted to represent a card, an enumeration could be used to represent the suit.

enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

Suit cardSuit;

cardSuit = Suit.CLUBS;

if(cardSuit == Suit.CLUBS){

System.out.println("The suit of this card is clubs.");

}

enumerations make code more readable and provide less room for programmer error.

## Understanding Variable Scope

Local variables

Method parameters

Instance variables

Static variables

## Local variables

are the variables that are declared inside of methods. The block

of code that a variable is declared in determines the scope of the local variable.

void sampleMethod() { // Start of code block A

int totalCount = 0;

for (int i = 0; i < 3; i++) { // Start of code block B

int forCount = 0;

totalCount++;

forCount++;

{ // Start of code block C

int block1Count = 0;

totalCount++;

forCount++;

block1Count++;

} // End of code block C

{ // Start of code block D

int block2Count = 0;

totalCount++;

forCount++;

block2Count++;

} // End of code block D

/\* These two variables have no relation to the above ones of

the same name \*/

int block1Count;

int block2Count;

} // End of code block B

} // End of code block A

When developing Java source code, variables should be declared with the

most limited scope possible. This is a coding practice that helps reduce

programming mistakes and improves code readability.

Method parameters are the variables that are passed to the method from the calling

segment of code.

Instance variables are the variables that are declared in the class. They are called

instance variables because they are created and remain in memory for as long asthe instance of the class exists. Instance variables store the state of the object. They

are not within the scope of any one particular method, instead they are in scope for

the entire class. They exist and retain their value from the time a class is initialized

until that class is either reinitialized or no longer referenced.

public class Television {

int channel = 0;

boolean on = false;

void setChannel(int channelValue) {

this.channel = channelValue;

}

int getChannel() {

return this.channel;

}

void setOn(boolean on) {

this.on = on;

}

boolean isOn() {

return this.on;

}

}

Television tv1 = new Television();

Television tv2 = new Television();

tv1.setChannel(2);

tv2.setChannel(7);

System.out.println("Television channel for tv1: " + tv1.getChannel());

System.out.println("Television channel for tv2: " + tv2.getChannel());

## Pass by value and pass by reference

Passing Primitives by Value to Methods

When a primitive is used as an argument, a copy of the value is made and given to

the method. If the method sets the value of the parameter to a different value, it has

no effect on the variable that was passed to the method. The following is an example

of a method that adds two to the int that is passed to it:

void addTwo(int value) {

System.out.println("Parameter: value = " + value);

value = value + 2;

System.out.println("Leaving method: value = " + value);

}

int value = 1;

System.out.println("Argument: value = " + value);

addTwo(value);

System.out.println("After method call: value = " + value);

Argument: value = 1

Parameter: value = 1

Leaving method: value = 3

After method call: value = 1

Objects are passed by reference to a method. This means that instead of making a

copy of the object and passing it, a reference to the original object is passed to the

method. A reference is basically an internal index that represents the object. Any actions that are performed

on the parameter will be reflected in the variable that was used as an argument.

public class Number {

int number;

public Number(int number) {

this.number = number;

}

int getNumber() {

return this.number;

}

void setNumber(int number) {

this.number = number;

}

}

void addThree(Number value) {

System.out.println("Parameter: value = " + value.getNumber());

value.setNumber(value.getNumber() + 3);

System.out.println("Leaving method: value = " + value.getNumber());

}

Number value = new Number(1);

System.out.println("Argument: value = " + value.getNumber());

addThree(value);

System.out.println("After method call: value = " + value.getNumber());

Argument: value = 1

Parameter: value = 1

Leaving method: value = 4

After method call: value = 4

## Packages-

Packaging is a common approach used to organize related classes and interfaces.

Most reusable code is packaged.

Packages are thought of as containers for classes, but actually they define where

classes will be located in the hierarchical directory structure. Packaging is

encouraged by Java coding standards to decrease the likelihood of classes colliding.

Packaging your classes also promotes code reuse, maintainability, and the objectoriented

principle of encapsulation and modularity.

Typically, larger packages support reusability, whereas smaller

packages support maintainability.

Maintainability Often, software changes can be limited to a single package

when the package houses focused functionality.

To place a source file into a package, use the

package statement at the beginning of that file. You may use zero or one package

statements per source file. To import classes from other packages into your source

file, use the import statement. The java.lang package that houses the core

language classes is imported by default.

package com.scjaexam.tutorial; // Package statement directory com/scjaexam/tutorial

/\* Imports class ArrayList from the java.util package \*/

import java.util.ArrayList;

/\* Imports all classes from the java.io package \*/

import java.io.\*;

public class MainClass {

public static void main(String[] args) {

/\* Creates console from java.io package \*/

Console console = System.console();

String planet = console.readLine("\nEnter your favorite

planet: ");

/\* Creates list for planets \*/

ArrayList planetList = new ArrayList();

planetList.add(planet); // Adds users input to the list

planetList.add("Gliese 581 c"); // Adds a string to the list

System.out.println("\nTwo cool planets: " + planetList);

}

}

$ Enter your favorite planet: Jupiter

$ Two cool planets: [Jupiter, Gliese 581 c

Static imports are a new feature to Java SE 5.0. Static imports allow you

to import static members. The following example statements would be valid in Java SE 5.0,

but would be invalid for J2SE 1.4.

/\* Import static member ITALY \*/

import static java.util.Locale.ITALY;

/\* Imports all static members in class Locale \*/

import static java.util.Locale.\*;