Contents

[Types of packages: 1](#_Toc161610691)

[Built-in Packages 1](#_Toc161610692)

[User-defined packages 2](#_Toc161610693)

[**U**sing Static Import 2](#_Toc161610694)

[Handling name conflicts 3](#_Toc161610695)

[Variable Arguments (Varargs) in Java 3](#_Toc161610696)

[Reference variable 5](#_Toc161610697)

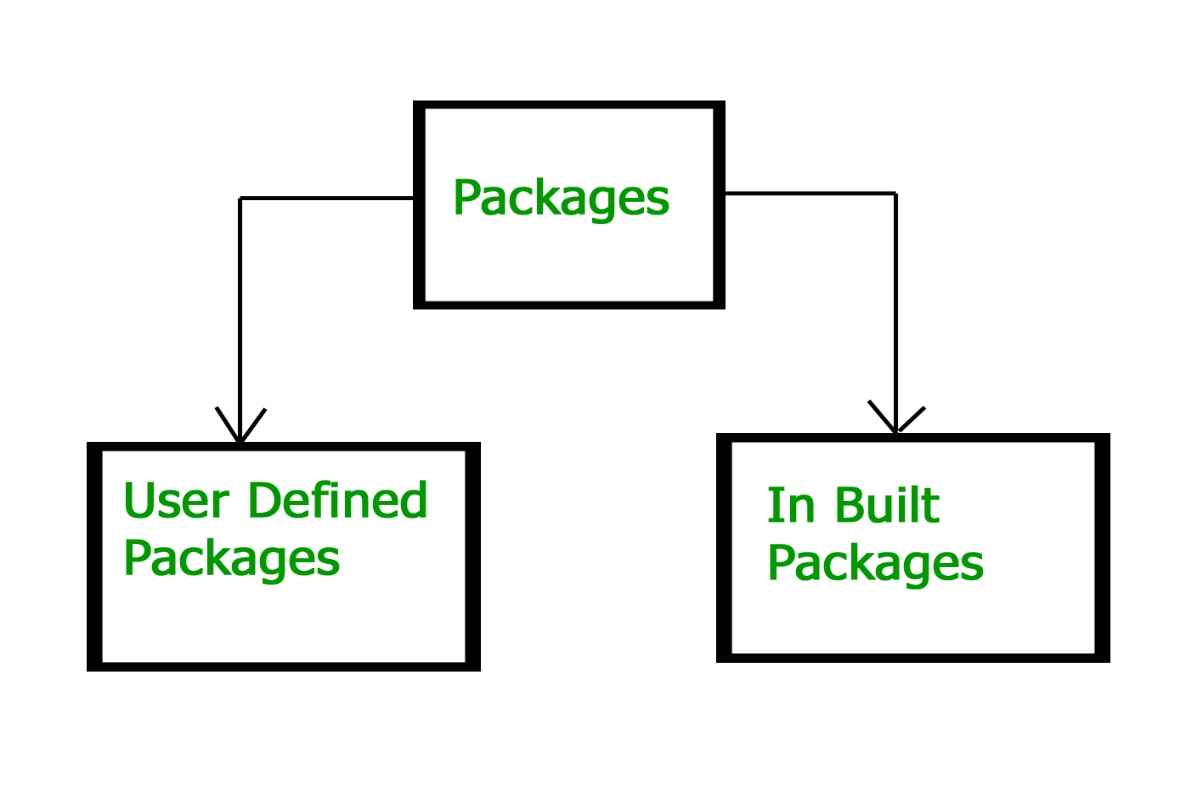
[What is "==" equality operator in Java 5](#_Toc161610698)

[1. What is equals method in Java? 6](#_Toc161610699)

[Summary 6](#_Toc161610700)

[Object toString() Method 6](#_Toc161610701)

# Types of packages:



Built-in Packages  
These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:  
1) **java.lang:**Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.  
2) **java.io:**Contains classed for supporting input / output operations.  
3) **java.util:**Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.  
4) **java.applet:**Contains classes for creating Applets.  
5) **java.awt:**Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).  
6) **java.net:**Contain classes for supporting networking operations.

User-defined packages  
These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**.

# Using Static Import

Static import is a feature introduced in Java programming language ( versions 5 and above ) that allows members ( fields and methods ) defined in a class as public static to be used in Java code without specifying the class in which the field is defined.  
Following program demonstrates static import :

|  |
| --- |
| // Note static keyword after import.  import static java.lang.System.\*;    class StaticImportDemo  {     public static void main(String args[])     {          // We don't need to use 'System.out'          // as imported using static.          out.println("GeeksforGeeks");     }  } Handling name conflicts The only time we need to pay attention to packages is when we have a name conflict . For example both, java.util and java.sql packages have a class named Date. So if we import both packages in program as follows:  import java.util.\*;  import java.sql.\*;  //And then use Date class, then we will get a compile-time error :  Date today ; //ERROR-- java.util.Date or java.sql.Date?  The compiler will not be able to figure out which Date class do we want. This problem can be solved by using a specific import statement:  import java.util.Date;  import java.sql.\*;  If we need both Date classes then, we need to use a full package name every time we declare a new object of that class. For Example:  java.util.Date deadLine = new java.util.Date();  java.sql.Date today = new java.sql.Date(); |

# Variable Arguments (Varargs) in Java

In JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called varargs and it is short-form for variable-length arguments. A method that takes a variable number of arguments is a varargs method.  
Prior to JDK 5, variable-length arguments could be handled two ways. One using overloaded method(one for each) and another put the arguments into an array, and then pass this array to the method. Both of them are potentially error-prone and require more code. The varargs feature offers a simpler, better option.

Syntax of varargs :  
A variable-length argument is specified by three periods(…). For Example,

public static void fun(int ... a)

{

// method body

}

This syntax tells the compiler that fun( ) can be called with zero or more arguments. As a result, here a is implicitly declared as an array of type int[]. Below is a code snippet for illustrating the above concept :

|  |
| --- |
| // Java program to demonstrate varargs  class Test1  {      // A method that takes variable number of integer      // arguments.      static void fun(int ...a)      {          System.out.println("Number of arguments: " + a.length);            // using for each loop to display contents of a          for (int i: a)              System.out.print(i + " ");          System.out.println();      }        // Driver code      public static void main(String args[])      {          // Calling the varargs method with different number          // of parameters          fun(100);         // one parameter          fun(1, 2, 3, 4);  // four parameters          fun();            // no parameter      }  } |

Note: A method can have variable length parameters with other parameters too, but one should ensure that there exists only one varargs parameter that should be written last in the parameter list of the method declaration.

We can overload vararg method but it may lead to ambiguity.

int nums(int a, float b, double … c)

In this case, the first two arguments are matched with the first two parameters and the remaining arguments belong to c.

# Reference variable

The only way you can access an object is through a reference variable. A reference variable is declared to be of a specific type and that type can never be changed. Reference variables can be declared as static variables, instance variables, method parameters, or local variables.

A reference variable that is declared as final can’t never be reassigned to refer to a different object. The data within the object can be modified, but the reference variable cannot be changed.

# What is "==" equality operator in Java

"==" or equality operator in Java is a binary operator provided by Java programming language and used to compare primitives and objects. In terms of comparing primitives like boolean, int, float "==" works fine but when it comes to comparing objects it creates confusion with equals method in Java.  
  
The equality operator or "==" compare two objects based on memory reference. so "==" operator will return true only if two object reference it is comparing represent exactly same object otherwise "==" will return false.

## 1. What is equals method in Java?

Equals() method is defined in Object class in Java and used for checking equality of two objects defined by business logic e.g. two Employees are considered equal if they have same empId etc.   
  
You can have your domain object and then [override equals method](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) for defining a condition on which two domain objects will be considered equal. equal has contracted with hashcode method in Java and whenever you override equals method you also need to [override hashcode() in Java](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html).   
  
Default implementation of equals provided in Object class is similar to "==" equality operator and return true if you are comparing two references to the same object.

Since equals() is a method defined in the Object class thus the default implementation of the equals() method compares the object references or the memory location where the objects are stored in the heap. Thus by default the equals() method checks the object by using the “==” operator  
  
It’s one of the Java best practice to override equals in Java to define equality based on business requirement.

### Summary

1) use == to compare primitive e.g. boolean, int, char etc, while use equals() to compare objects in Java.

2) == return true if two reference are of same object. Result of equals() method depends on overridden implementation.

3) For comparing String use equals() instead of  == equality operator.

# Object toString() Method

Every class in java is child of Object class either directly or indirectly. Object class contains toString() method. We can use toString() method to get string representation of an object. Whenever we try to print the Object reference then internally toString() method is invoked. If we did not define toString() method in your class then Object class toString() method is invoked otherwise our implemented/Overridden toString() method will be called.

**NOTE:**In all wrapper classes, all collection classes, String class, StringBuffer, StringBuilder classes toString() method is overriden for meaningful String representation. Hence, it is highly recommended to override toString() method in our class also

**Java equals() and hashCode()**

Java equals() and hashCode() methods are present in Object class. So every java class gets the default implementation of equals() and hashCode()

According to java documentation of equals() method, any implementation should adhere to following principles.

* For any object x, x.equals(x) should return true.
* For any two object x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
* For multiple objects x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
* Multiple invocations of x.equals(y) should return same result, unless any of the object properties is modified that is being used in the equals() method implementation.
* Object class equals() method implementation returns true only when both the references are pointing to same object.

[**Java hashCode()**](https://www.digitalocean.com/community/tutorials/java-equals-hashcode#java-hashcode)

Java Object hashCode() is a native method and returns the integer hash code value of the object. The general contract of hashCode() method is:

* Multiple invocations of hashCode() should return the same integer value, unless the object property is modified that is being used in the equals() method.
* An object hash code value can change in multiple executions of the same application.
* If two objects are equal according to equals() method, then their hash code must be same.
* If two objects are unequal according to equals() method, their hash code are not required to be different. Their hash code value may or may-not be equal.
* If o1.equals(o2), then o1.hashCode() == o2.hashCode() should always be true.
* If o1.hashCode() == o2.hashCode is true, it doesn’t mean that o1.equals(o2) will be true.

[**When to override equals() and hashCode() methods?**](https://www.digitalocean.com/community/tutorials/java-equals-hashcode#when-to-override-equals-and-hashcode-methods)

When we override equals() method, it’s almost necessary to override the hashCode() method too so that their contract is not violated by our implementation. Note that your program will not throw any exceptions if the equals() and hashCode() contract is violated, if you are not planning to use the class as Hash table key, then it will not create any problem. If you are planning to use a class as Hash table key, then it’s must to override both equals() and hashCode() methods.

<https://www.digitalocean.com/community/tutorials/java-equals-hashcode>