

Packages and Import statements

Packages

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer.

Packages are important for three main reasons

1. They help in the overall organization of a project or library.
2. Packages give you a name-scoping, to help to prevent collisions if many programmers in a company decide to make a class with the same name.
3. Packages provide a level of security, because you can restrict the code, which you write so that only other classes in the same package can access it.

Creating a package

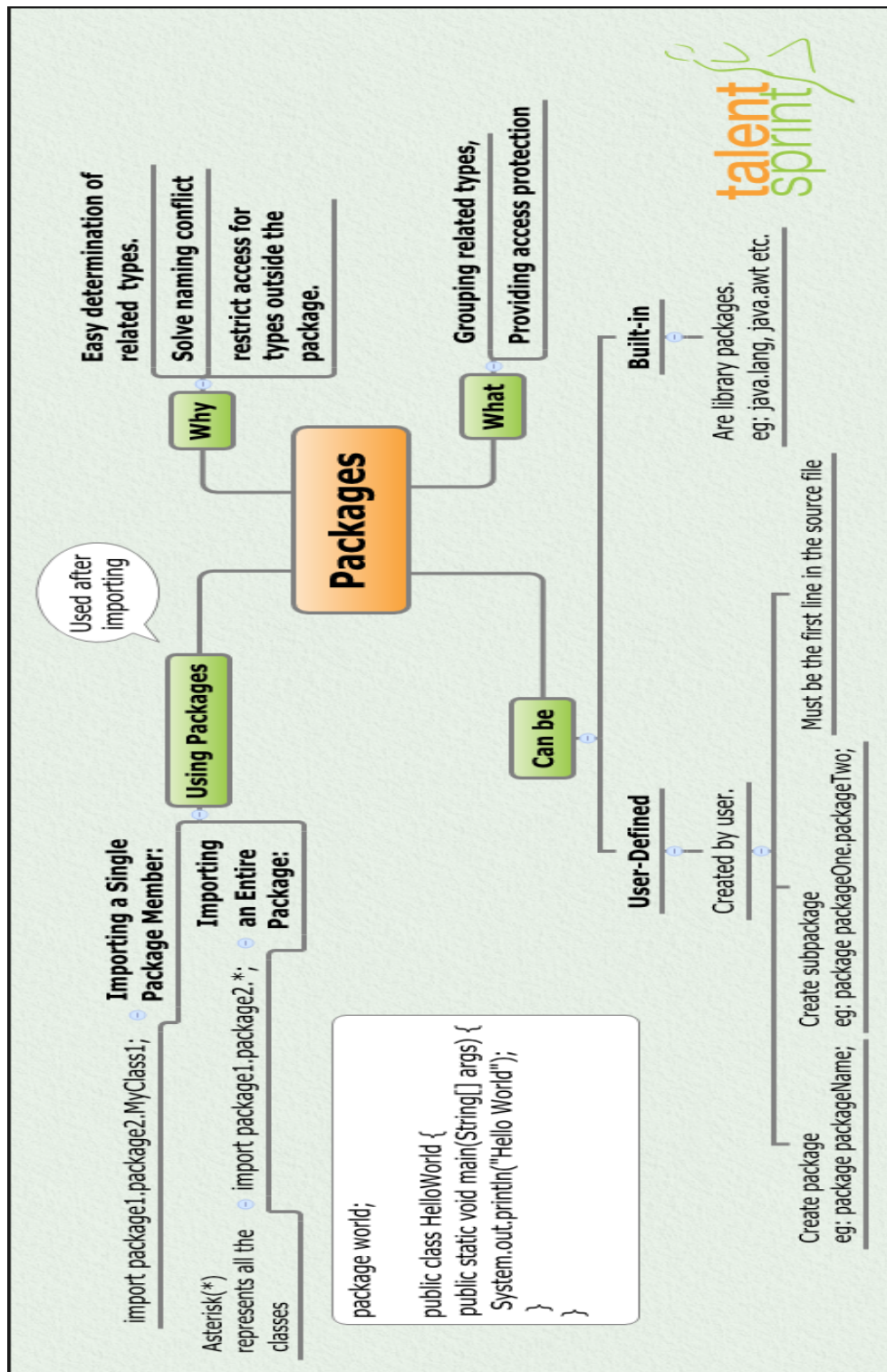
- Package statement should be the first non-comment line in the source code.
- Names of the package should be in lowercase.
- Package can have sub-package and are separated by a period separator (.)

```
1 package com.pack;  
2 public class MyClass {  
3     // statement(s);  
4 }
```

- Compile the source file and execute as:

```
$ javac -d . MyClass.java  
$ java com.talentsprint.MyClass
```

- A class must be in a directory structure that exactly matches the package structure. For a class, *com.talentsprint.MyClass*, the *MyClass* class must be in a directory named *talentsprint*, which is in a directory named *com*.



Import Statement

The import statements must come right after the package statement, before the class statement.

Purpose of import statement

- The rigid source code naming convention, the Java compiler can easily find the corresponding source or class files just from the fully qualified name of a package and class.

```

1  import java. util . ArrayList ;
2  public class MyClass {
3      public static void main (String [] args) {
4          ArrayList myList = new ArrayList(50);
5          // Statement(s);
6      }
7  }
```

- If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package can find each other without any special syntax.

Access Modifiers

Access modifiers defines the scope of a variable or method or class. There are four different types of access modifiers in Java:

1. public (Least restrictive)
2. protected
3. default
4. private (Most restrictive)

Public specifies that class members (variables or methods) are accessible to anyone, both inside and outside of the class or package. Any object that interacts with the class can have access to the public members of the class

Protected specifies that the class members are accessible only to methods in that class and the subclasses of the class. The subclass can be in different packages.

Default specifies that only classes in the same package can have access to the variables and methods of the class. No actual keyword is used for the default modifier and it is applied in the absence of an access modifier.

Private specifies that the class members are only accessible by the class in which they are defined.

Below is the sample code snippet that demonstrates how to use access specifiers

```

1  public class Student {
2      public String name; // public member
3      protected int totalMarks; // protected member
4      private double totalFeePaid; // private member
5      int age; // default member
6  }
```

Understanding from the above sample code snippet:

name property/variable can access by any class member from anywhere.

totalMarks property/variable can access by the class members itself and by its derived class members.

totalFeePaid property/variable can access by the class members where it is defined. No other class or class members can access.

age property/variable can access by any class which resides in the same package the class is defined.

Java Program Structure: The Access Modifiers

	private	default	protected	public
Same class	yes	yes	yes	yes
Same package		yes	yes	yes
Different package (sub-class)			yes	yes
Different package (non sub-class)				yes

Methods

A Java **method** is a collection of statements that are grouped together to perform an operation and are used to communicate with objects. When we invoke or call a method we are asking the object to carry out a task, thus can say methods implement the behaviour of objects. Methods are similar to the functions in **C** - language. Every method needs to have name, required input parameters, return type and set its visibility (private, protected or public).

Defining method

A method is just a chunk of code that does a particular job. The method structure contains method header and method body. The header contains return type, if any, a method name to identify the method and parameter(s) to pass values to the method. The body contains statements to perform the job.

```
<modifier> <returnType> <methodName> (<parametersList>) {  
    statement(s);  
}
```

Here is an example for a method named 'getTotalMarks' that takes 3 parameters of marks and return the total of the marks.

```
1  /** The snippet returns the total of 3 marks */  
2  public int getTotalMarks(int marks1, int marks2, int marks3) {  
3      return marks1 + marks2 + marks3;  
4  }
```

Method Calling

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when:

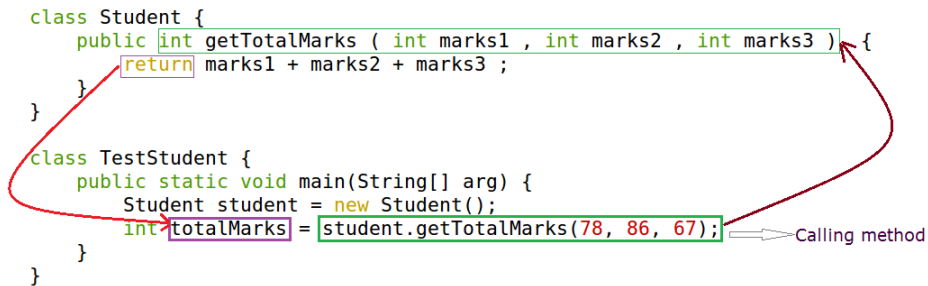
1. return statement is executed.
2. reaches to the end of the method.

```

class Student {
    public int getTotalMarks ( int marks1 , int marks2 , int marks3 ) {
        return marks1 + marks2 + marks3 ;
    }
}

class TestStudent {
    public static void main(String[] arg) {
        Student student = new Student();
        int totalMarks = student.getTotalMarks(78, 86, 67);
    }
}

```



Constructors

Java constructors are the methods which are used to initialize objects. Constructors are called implicitly when an object is instantiated.

Keypoints to remember:

- Constructor has the same name as the name of the class to which it belongs.
- Constructor is called or invoked when an object of class is created and can't be called explicitly.
- Constructor is generally declared as public.
- Constructor may have optional list of arguments.
- Constructor does not have any return type and not even void, since constructors never return a value.
- Constructor cannot invoke on an existing object.
- Constructor is used only in combination with the **new** operator, which is used to instantiate an object.

To declare a constructor, you write,

```

<modifier> <className> (<parametersList>) {
    statement(s);
}

```

Constructors are two types

Default Constructor is a constructor that doesn't have any parameters. If the class does not specify any constructors, then an implicit default constructor is created

Syntax for default constructor

```
class <ClassName> {
    <ClassName>() {
        // Statement(s);
    }
}
```

Parameterized constructor is a constructor that has parameters. These are required to pass parameters on creation of objects. Parameterized constructor is used to provide different values to the distinct objects.

Syntax for parameterized constructor

```
class <ClassName> {
    <ClassName>(<parametersList>) {
        // Statement(s);
    }
}
```

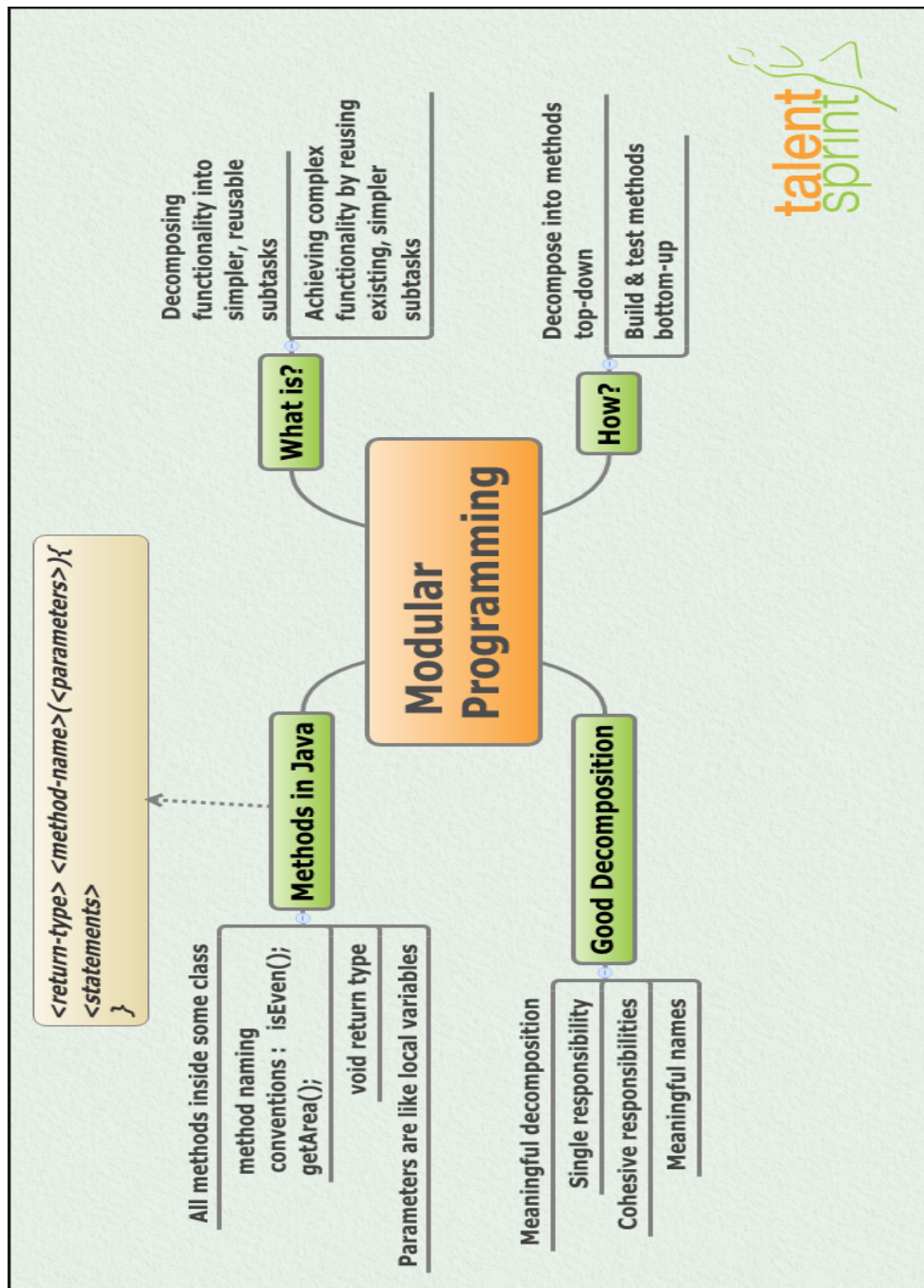
Invoking Constructors

Constructors are invoked while creating an object using **new** operator.

```
1  class Student {
2      // Instance variables
3      // Default constructor
4      Student() {
5          // Statement(s);
6      }
7
8      // Parameterized constructor
9      Student(String name, String branch) {
10         // Statement(s);
11     }
12
13     public static void main( String [] args ){
```

Advanced Class Design

```
14      //create objects for Student class
15      // Invokes default constructor
16      Student = new Student();
17      // Invokes parameterized constructor
18      Student studentCris = new Student("Surya", "CSE");
19      //some code here
20  }
21 }
```

this keyword

this is a keyword in Java. Which can be used inside method or constructor of class. It works as a reference to current object whose method or constructor is being invoked. **this** keyword can be used to refer any member of current object from within an instance method or a constructor.

this keyword with constructor (Constructor chaining)

- Constructor calls can be chained, which means that one constructor can call another constructor. **this()** method is used for this purpose.
- There are a few things to remember when using *this()* method in constructor call
 - When using this in constructor call, it must occur as the first statement.
 - It can only be used in a constructor definition. The this call can then be followed by any other relevant statements.

```

1  class Student {
2      public Student(){
3          this("some string");
4      }
5      public Student(String str){
6          // Statement(s);
7      }
8      public static void main( String [] args ) {
9          Student student = new Student();
10     }
11 }
12

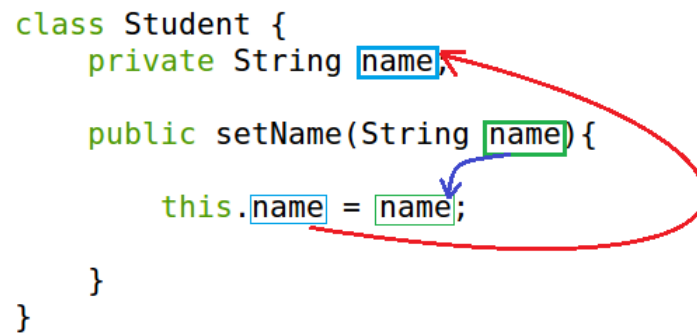
```

this keyword with field (Instance Variable)

- **this** keyword can be very useful in case of Variable Hiding.
- It(**this**) works as a reference to current object whose method or constructor is being invoked.
- To use this reference, you type, **this**.<Variable—Name >

```
class Student {
    private String name;

    public setName(String name) {
        this.name = name;
    }
}
```



Final keyword

final keyword in java is used to restrict the user. Final is a *keyword* or *reserved* word in java and can be applied to member variables, methods, class and local variables in Java. Once you make a reference final you are not allowed to change that reference and compiler will verify this and raise compilation error if you try to re-initialized final variables in java.

Final classes

A **final** class cannot be subclassed or inherited. Several classes in Java are final. e.g. java.lang.String, java.lang.System, Integer and other wrapper classes.

Let us observe the following code snippet, where 'Student' class inherits a **final** class 'Person'

```
final class Person {
    // Instance variables
    // Instance methods
}

class Student extends Person {
    // Instance variables
    // Instance methods
}
```

When you try to compile the above program, it display an compile time error, *"cannot inherit from final class"*

Final Methods

A **final** method cannot be overridden or hidden by subclasses. This is used to prevent unexpected behavior from a subclass altering a method that may be crucial to the function or consistency of the class.

```
1  class Person {
2      public void eats() {
3          // Statement(s);
4      }
5      public final void dateOfBirth() {
6          // Statement(s);
7      }
8  }
9
10 class Student extends Person {
11     // Ok, overriding Person#eats()
12     public void eats() {
13         // Statement(s);
14     }
15
16     // forbidden
17     public void dateOfBirth() {
18         // Statement(s);
19     }
20 }
```

Final Variables

A **final** variable can only be initialized once, either via an initializer or an assignment statement. **final** variable does not need to be initialized at the point of declaration: this is called a “blank final” variable.

- A blank final instance variable of a class must be definitely assigned in every constructor of the class in which it is declared
- Similarly, a blank final static variable must be definitely assigned in a static initializer of the class in which it is declared

```

1  class Student {
2
3      public final String name;
4      public final int age;
5      public final String branch;
6      Sphere(String n, int a, String b) {
7          name = n;
8          age = a;
9          branch = b;
10     }
11
12     // Some methods
13 }
14

```

Benefits of final keyword in Java

- Final keyword improves performance. Not just JVM can cache final variable but also application can cache frequently use final variables.
- Final variables are safe to share in multi-threading environment without additional synchronization overhead.
- Final keyword allows JVM to optimize method, variable or class.

Static keyword

static keyword in Java is used for memory management mainly. It can be applied to a field, a method or an inner class. A static field, method or class has a single instance for the whole class that defines it, even if there is no instance of this class in the program.

Note: For instance, a Java entry point (main()) has to be static.

```
public static void main(String args [])
```

static method

A **static** method cannot be **abstract**. It must be placed before the variable type or the method **return** type. It is recommended to place it after the access modifier and before the

final keyword.

Assume 'getTotalMarks' method is final in 'Student' class, then the method is defined as below code snippet

```

1  class Student {
2      // Some variables and methods
3      public static int getTotalMarks(Student [] stArray) {
4          double total = 0.0;
5          //Statement(s);
6          return total ;
7      }
8  }
9

```

static methods in Java can be called without creating an object of class.

```

1  class TestStudent {
2      public static void main(String [] args) {
3          int totalMarks = Student.getTotalMarks();
4      }
5  }

```

Keypoints to remember

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.
- The static method can not use non static data member or call non-static method directly.
- this and super cannot be used in static context.

static variables

A **static** variable can be used as data sharing amongst objects of the same class. Mostly the **static** variable is declared as **public**. **static** variables are common for each object of that class and there is only one instance of it.

```
1  class Person {  
2      static String name;  
3  }  
4
```

static block

A **static** block is used to initialize the static data member and is executed before main method at the time of classloading.

```
1  public class StaticBlock{  
2      static {  
3          System.out.println (" static block is invoked");  
4      }  
5      public static void main(String args []) {  
6          System.out.println (" Hello main");  
7      }  
8  }
```

Output

```
$ javac StaticBlock.java  
$ java StaticBlock  
static block is invoked  
Hello main  
$ |
```

Garbage Collector

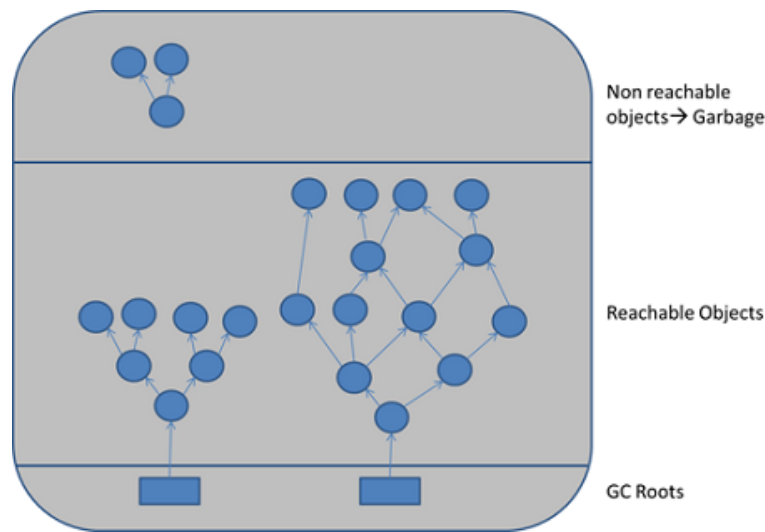
Garbage collection is the systematic recovery of pooled computer storage that is being used by a program when that program no longer needs the storage. This frees the storage for use by other programs (or processes within a program). Garbage collection is an automatic memory management feature in many modern programming languages, such as Java and languages in the .NET framework. Languages that use garbage collection are often interpreted or run within a virtual machine like the JVM. In each case, the environment that runs the code is also responsible for garbage collection. Programs with an automatic garbage collector (GC) try to eliminate bugs by automatically detecting when a piece of data is no longer needed.

The Garbage Collector runs in a thread in the JVM. When the free memory drops under a threshold the GC begins to run to eliminate unneeded objects. Exactly when it runs and for how long, however, cannot be controlled by the user program.

Garbage-Collection Roots

Every object tree must have one or more root objects. As long as the application can reach those roots, the whole tree is reachable. There are four kinds of GC roots in Java

1. **Local variables** are kept alive by the stack of a thread. This is not a real object virtual reference and thus is not visible. For all intents and purposes, local variables are GC roots.
2. **Active Java threads** are always considered live objects and are therefore GC roots. This is especially important for thread local variables.
3. **Static variables** are referenced by their classes. This fact makes them de facto GC roots. Classes themselves can be garbage-collected, which would remove all referenced static variables. This is of special importance when we use application servers, OSGi containers or class loaders in general. We will discuss the related problems in the Problem Patterns section.
4. **JNI References** are Java objects that the native code has created as part of a JNI call. Objects thus created are treated specially because the JVM does not know if it is being referenced by the native code or not. Such objects represent a very special form of GC root, which we will examine in more detail in the Problem Patterns section below.



Working with Garbage Collection

Java garbage collection is an automatic process to manage the runtime memory used by programs. By doing it automatic JVM relieves the programmer of the overhead of assigning and freeing up memory resources in a program.

Garbage Collection GC Initiation

Being an automatic process, programmers need not initiate the garbage collection process explicitly in the code. `System.gc()` and `Runtime.gc()` are hooks to request the JVM to initiate the garbage collection process.

`finalize()` method

Sometime an object will need to perform some specific task before it is destroyed such as closing an open connection or releasing any resources held. To handle such situation `finalize ()` method is used. `finalize ()` method is called by garbage collection thread before collecting object. Its the last chance for any object to perform cleanup utility.

The below code snippet shows how garbage collector is called explicitly using `System.gc()`

```
1 import java.lang.*;
2 public class SystemGCDemo {
3     public static void main(String[] args) {
4         int arr1 [] = { 0, 1, 2, 3, 4, 5 };
5     }
6 }
```

```

5      int arr2 [] = { 0, 10, 20, 30, 40, 50 };
6      // copies an array from the specified source array
7      System.arraycopy(arr1, 0, arr2, 0, 1);
8      System.out. print (" array2 = ");
9      System.out. print (arr2 [0] + " ");
10     System.out. println (arr2 [1] + " ");
11     // it runs the GarbageCollector
12     System.gc();
13     System.out. println (" Cleanup completed... ");
14 }
15 }

```

The below code snippet shows how garbage collector is called explicitly using Runtime.
getRuntime().gc();

```

1  public class RuntimeGCTest {
2      public static void main(String[] args) throws InterruptedException {
3          ColorCode colorCode = new ColorCode("white");
4          colorCode = null;
5          Runtime.getRuntime().gc();
6      }
7  }
8
9  class ColorCode {
10     private String color;
11     public ColorCode(String color) {
12         this. color = color;
13     }
14
15     @Override
16     public void finalize () {
17         System.out. println (this. color + " cleaned");
18     }
19 }

```

The below code snippet shows that the garbage collection automatically works behind without System.gc() called.

```

1  public class GCTest {

```

```

2      public static void main(String[] args) throws InterruptedException {
3          ColorCode colorCode = new ColorCode("white");
4          for (int i = 0; i < 100000000; i++) {
5              if (i % 2 == 1) {
6                  colorCode = new ColorCode("red");
7              } else {
8                  colorCode = new ColorCode("green");
9              }
10             colorCode = null;
11         }
12     }
13 }
14
15 class ColorCode {
16     private String color;
17     public ColorCode(String color) {
18         this.color = color;
19     }
20
21     @Override
22     public void finalize () {
23         System.out.println (this.color + " cleaned");
24     }
25 }

```

What does a GC perform?

- If a section of memory earlier used by an object is no longer referenced by any variable, then the Garbage Collector (GC) will release this memory for re-use by new objects. This takes place automatically without any input from the user program.
- There are no heap commands such malloc() and free() functions in C needed to allocate or de-allocate memory buffers.
- Java does use a similar "new" operator as in C++ for creating objects but has no "delete" operator.