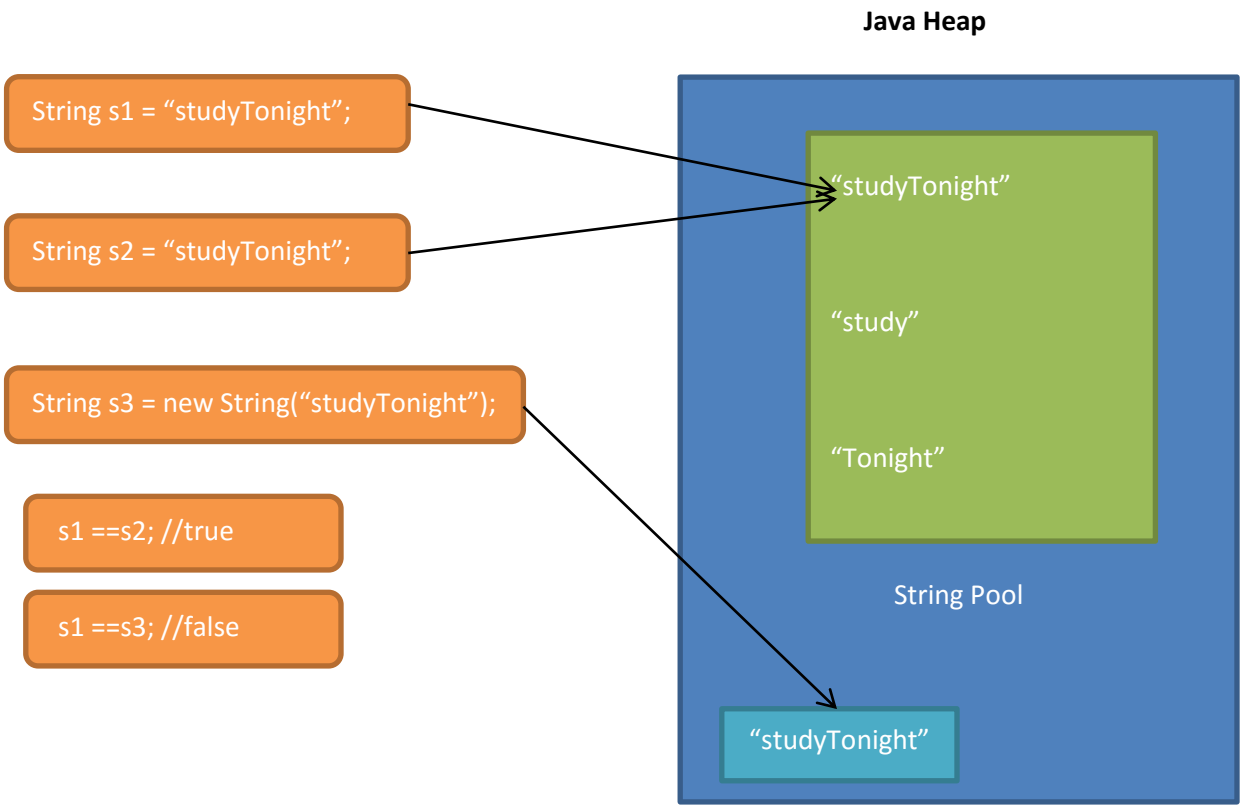
String Pool

As the name suggest, String pool is the "pool of String". Let's take an example and we will see how String pool works in java.

We all know that we can create a string in java using double quotes. Here we will create 3 Strings references s1, s2 and s3.

String s1 = "studyTonight";
String s2 = "studyTonight";
String s3 = new String("studyTonight");

When we use double quotes to create a String, it first looks for String with same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference. However using *new* operator, we force String class to create a new String object.



Here, in our example java will first search for "studyTonight" in String pool and it will not find for the first time and it will create a new object and returns the reference to s1. Now, for the second time it will find an object with same value in String pool, so it will return the same object to s2 reference. But while using new operator we will force Java to create a new object.

Why we need String Pool?

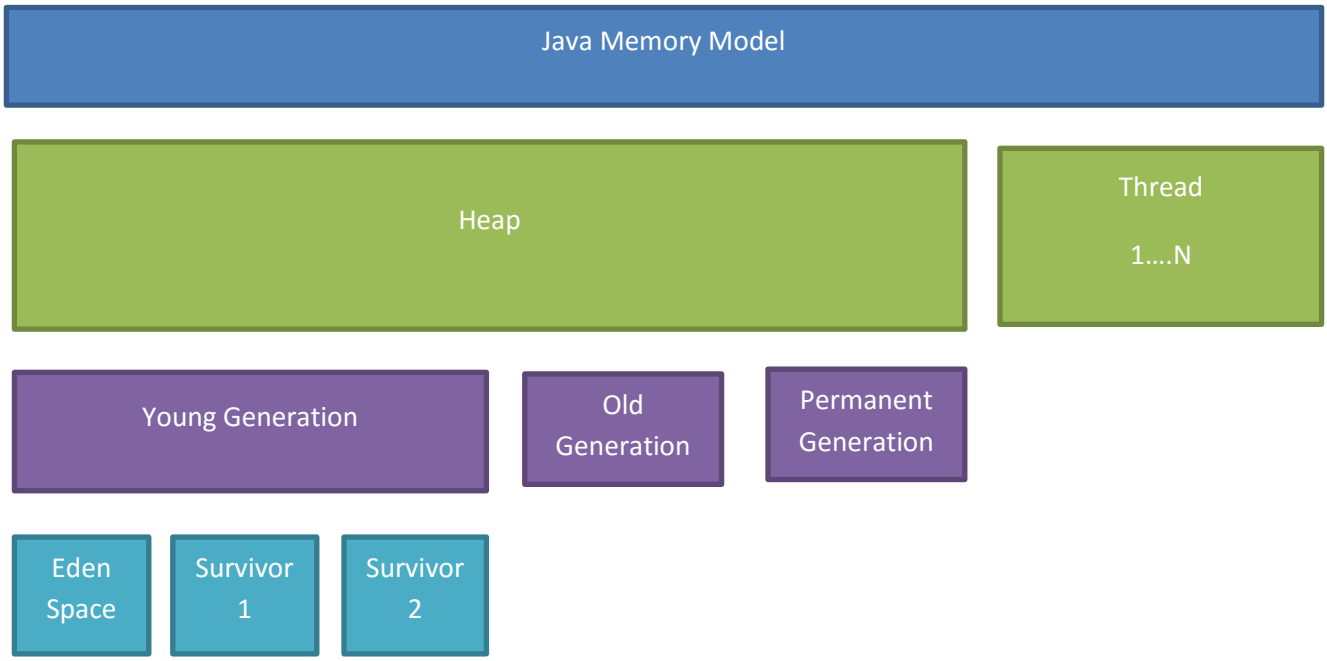
The simplest answer to this question is to limit the increasing size of String objects in java memory. With the use of string pool we can reduce the object creation for similar value objects and use the existing object created in string pool.

Quick Fact:

1. String pool is inside PermGen space of the heap area of Java memory whereas normal string object creation happens in young generation of Heap are of Java memory.
2. Use of intern() method on String will give the reference to the object in String pool, if object is already present, otherwise it creates a new object in String pool and gives the reference to it.
3. The creation of two strings with the same sequence of letters without the use of the *new* keyword will create pointers to the same String in the Java String literal pool.

```
String s4 = "study"+ "Tonight";
String s5 = "stu" + "dyTonight";
S5==s4; //True
```

Java Memory Model



Java memory is majorly divided in 2 regions: **Heap** and **Stack (Thread)**

Java Heap:
Java Heap is the area where objects are created. Java Heap is divided into three main regions named as New or Young Generation, Old or Tenured Generation and Perm space.

Young Generation:

Young generation, also called as new generation, is the place where all the new objects are created.

Young Generation is divided into three parts – **Eden Space** and two **Survivor Memory** spaces.

When Eden space is filled with objects, garbage collection is performed and all the survivor objects are moved to one of the survivor spaces. This garbage collection is called **Minor GC.** Objects that are survived after many cycles of minor GC, are moved to the Old generation memory space.

Old Generation:
Old Generation, also called as tenure generation, contains the objects that are long lived and survived after many rounds of Minor GC. Usually garbage collection is performed in Old Generation memory when it's full.

Permanent Generation:

Permanent Generation, also called as PermGen, Space is the space of Java Heap is where JVM stores Meta data about classes and methods (called as Method Area), String pool (also called as Memory Pool) and Class level details.

Java Stack:

Java Stack is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method.

Static in Java

The concept of static is used whenever data which is common to the objects of the class is required.

Static is keyword in java which can be applied to following elements:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

Static Variable:

- If you declare any variable as static, it is known static variable.
- Static variables are also called as class variables and they are different from instance variables of the class.
- Static members represent data that is common to the entire class e.g. company name of employees, college name of students etc.
- A single copy of the data will be shared by all instances of the class
- They are initialized at class load time. If not explicitly initialized during declaration, they will have default values
- Unlike local variables, static variables are stored in heap, because they are common to all object of a class.
- Syntax for declaring static variable

<<Access specifier>><<static>> <<datatype>> <<variable name>>

Example:

```
class Student{
   String name;
   static String college ="StudyTonight";

   Student8(String n){
   name = n;
   }
 void display (){
     System.out.println(name+" "+college);
}
 public static void main(String args[]){
 Student s1 = new Student("Manan");
 Student s2 = new Student("Piyush");

 s1.display();
 s2.display();
 }
}
```

(Code Below)

```
class Student{
  String name;
  static String college ="StudyTonight";

  Student(String n){
  name = n;
  }
 void display (){
          System.out.println(name+" "+college);
}
 public static void main(String args[]){
 Student s1 = new Student("Manan");
 Student s2 = new Student("Piyush");

 s1.display();
 s2.display();
 }
}
```

Output:

Manan StudyTonight

Piyush StudyTonight

Static Method:

- Static method is a method that is generic to the entire class.
- It may be used for accessing the static variables
- Syntax for declaring static method:

<<Access Specifier>> <<static>><<return datatype>><<method name>> {

}

- Static methods are invoked using the syntax: classname.methodname();
- For invoking static method or variables, an object is not required. However an object can also be used to access the static method/variable.
- Static methods can access only other static data members and methods.
- Non static variable cannot be accessed directly inside a static method.

Example:

```
class Student{
   String name;
   static String college ="StudyTonight";

    static void change(){
     college = "StudyToday";
     }

   Student(String n){
   name = n;
   }
 void display (){
     System.out.println(name+" "+college);
}
 public static void main(String args[]){

 Student.change();
 Student s1 = new Student("Manan");
 Student s2 = new Student("Piyush");

 s1.display();
 s2.display();
 }
}
```

(code below)

class Student{

String name;

static String college ="StudyTonight";

```
   static void change(){
    college = "StudyToday";
    }

  Student(String n){
  name = n;
  }
 void display (){
            System.out.println(name+" "+college);
}
 public static void main(String args[]){

 Student,change();
 Student s1 = new Student("Manan");
 Student s2 = new Student("Piyush");

 s1.display();
 s2.display();
 }
}
```

Output:

Manan StudyToday

Piyush StudyToday

Static Block:

- The static block is a block of statement inside a Java class which gets executed when a class is first loaded in memory
- A static block helps to initialize the static data members just like how constructors help to initialize instance members

- A class can have any number of static blocks
- The execution will be based on the order in which the blocks are written

Example:

```
class StdyTonight{
  static{
System.out.println("This is static block");
}
  public static void main(String args[]){
   System.out.println("StudyTonight");
  }
}
```

Static (Nested) Class:

Java allows you to define a class within another class. Such a class is called a nested class. Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are called static nested classes.

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation.
- It can lead to more readable and maintainable code

Example:

```
class OuterClass {
   ...
   static class StaticNestedClass {
      ...
   }
   class InnerClass {
      ...
   }
}
```

Annotations

Annotations, a form of metadata, provide data or information to the compiler about a program that is not part of the program itself. This information can be anything like suppress warnings, generate code or some runtime information.

The at sign character (@) indicates to the compiler that following is an annotation.

E.g. @Override

Annotation can also contain elements.

E.g. @SuppressWarnings(value = "unchecked")

Predefined Annotation Types:

These are the annotations which are already defined in java. They are @Deprecated, @Override, and @SuppressWarnings.

@Deprecated: This annotation indicates that the marked element is deprecated and must not be used. The compiler generates a warning whenever a program uses a method, class, or field with the @Deprecated annotation.

E.g. Below is the example of deprecated method in java.

```
  @Deprecated
  static void deprecatedExample() { }
}
```

@Override: This annotation informs the compiler that the element is overridden with an element declared in a superclass.

E.g. Below is the example of overridden method in java.

```
  @Override
  int overriddenExample() { }
```

@SuppressWarnings: This annotation type allows Java programmers to suppress (disable) compilation warnings for a certain part of a program like field, method, parameter, constructor, and local variable.

```
@SuppressWarnings("unchecked")

void uncheckedExample() {
```

```
   List words = new ArrayList();



   words.add("studyTonight"); //Line -4

}
```

If we do not use @ SuppressWarnings annotation then, it will show warning for above line 4.

Custom Annotations:
In java, we can create our own annotation using @interface annotation.
This is very similar to interface in java. Here we have to define data type and name of the element.
E.g.

```
@interface CustomAnnotation {
  String   name();
   int  age();
}
```

To use this annotation, we can code like below:

```
@CustomAnnotation(
   name="Manan",
   age=25,
   )
public class CustomClass {



}
```

Annotations for Annotations
Annotations that apply to other annotations are called meta-annotations.

@Retention: This annotation specifies how the marked annotation is stored:
RetentionPolicy.SOURCE – The marked annotation is retained only in the source level and is ignored by the compiler.
RetentionPolicy.CLASS – The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).
RetentionPolicy.RUNTIME – The marked annotation is retained by the JVM so it can be used by the runtime environment.

@Documented: This annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool.

@Target: This annotation is used to restrict the types of Java elements that the particular annotation can be applied. Following are the element types as its value:

ElementType.ANNOTATION_TYPE can be applied to an annotation type.
ElementType.CONSTRUCTOR can be applied to a constructor.
ElementType.FIELD can be applied to a field or property.
ElementType.LOCAL_VARIABLE can be applied to a local variable.
ElementType.METHOD can be applied to a method-level annotation.
ElementType.PACKAGE can be applied to a package declaration.
ElementType.PARAMETER can be applied to the parameters of a method.
ElementType.TYPE can be applied to any element of a class.

@Inherited: This annotation indicates that the annotation type can be inherited from the super class, which is not true by default.

@Repeatable: This annotation indicates that the marked annotation can be applied more than once.

E.g.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
@Repeatable
public @interface CustomAnnotation {
…
}
```