JAVA @11

Final

Objective

After completing this session you will be able to understand,

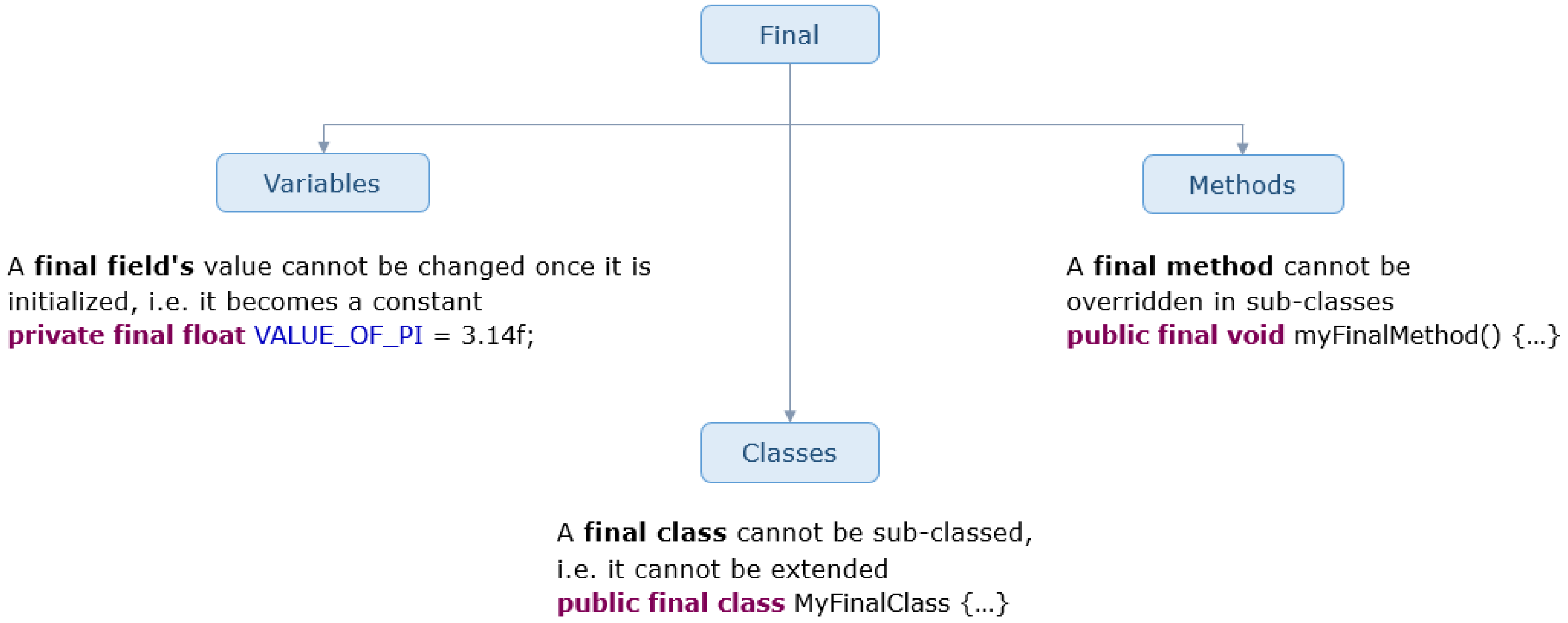
- Understand final keyword
- final Variables
- final Methods
- final Classes

final - Keyword

Remember how we used π (Pi) to calculate surface areas and volumes? It is a constant with the value 3.14.

While programming, you may come across various situations where you would have to create components which should never change, i.e. remain constant. This is where the final keyword comes into picture.

This keyword can be used with variables, classes and methods.



A **final field's** value cannot be changed once it is initialized, i.e. it becomes a constant

```
private final float VALUE_OF_PI = 3.14f;
```

A **final method** cannot be overridden in sub-classes

```
public final void myFinalMethod() {...}
```

A **final class** cannot be sub-classed, i.e. it cannot be extended

```
public final class MyFinalClass {...}
```

Final Variable

The code given below demonstrates the use of final variables that are initialized while declaring.

It also illustrates how these unassigned final variables and static final variables can be assigned a value.

Execute the code and observe the output.

```java
class Movie{
    final String name="The Godfather";
    void watchMovie(){
        //name="Chinatown";
        //Uncomment the above line
        //This throws a compilation error saying "final field 'movies' cannot be reassigned"
        System.out.println("Movie Name: "+name);
    }

    final int shows;  // (blank/unassigned) final variable
    public Movie(){
        shows=400;
        //The unassigned final variable can be assigned a value only in the constructor
        System.out.println("Total shows: "+shows);
    }

    static final float price;  // static blank final variable
    static{
        price=200.25f;
        // The unassigned static final variable can be assigned a value only in a static block
        System.out.println("Price: "+Movie.price);
    }
}
public class FinalVariables {
    public static void main(String[] args) {
        Movie movie=new Movie();
        movie.watchMovie();
    }
}
```

Final Methods

The code given below demonstrates the use of the final method and it has a compilation error.

Understand the working and try to find out the possible ways to resolve the compilation error from the code. Line no: 9

```
1. package com.rntbci.learning.finalkey;
2. class Android{
3.     final void ringtone() {
4.     System.out.println("Mobile phone is ringing.");
5.     }
6. }
7. class Samsung extends Android{
8.     @Override
9.     void ringtone() {
10.        System.out.println("Samsung is ringing.");
11.    }
12.}
13.public class FinalMethods {
14.    public static void main(String[] args) {
15.        //creating the object of Android
16.        Android obj1=new Android();
17.        obj1.ringtone(); //It will invoke the method in the parent class
18.        Samsung obj2=new Samsung();
19.        obj2.ringtone();
20.        //Cannot invoke ringtone() from child class as the
21.        //parent class has marked it final.
22.    }
23.}
```

Final Class

The code given below demonstrates the use of final class.

Observe what happens when a child class PermanentEmployee is created for final parent class Employee.

```java
package com.rntbci.learning.finalkey;

public class FinalClass {
   public static void main(String[] args) {
      Employee e=new Employee();
      PermanentEmployee pe=new PermanentEmployee();
   }
}

final class Employee{
   Employee() {
     System.out.println("Inside Final Parent Constructor");
  }
}

class PermanentEmployee extends Employee{
   //creating child class for final parent class
   PermanentEmployee() {
      System.out.println("Inside child of Final Parent constructor.");
   }
}
/*
 * This class will throw compilation error saying
 * "PermanentEmployee cannot subclass the final class Employee"
*/
```

Thank you

You have successfully completed

Java Final keyword

JAVA @11

Memory Management

Objective

After completing this session you will be able to understand,

- Memory Allocation
- Memory De-Allocation
- Garbage Collection

Memory allocation Scenario

Till now you have learnt the basic concepts of OOP but have you wondered how Java manages object creation in the memory?

Consider the Customer class.

```java
public class Customer {
    public String customerId;
    public String customerName;
    public long contactNo;
    public String address;
    public Customer(String customerId, String customerName, long contactNo,
String  address) {
        this.customerId = customerId;
        this.customerName = customerName;
        this.contactNo = contactNo;
        this.address = address;
    }
}
```

What do you think will happen when the following statement gets executed?

```java
Customer customerObj = new Customer("C101", "Stephen", 7856341287L,
"D089, Louis Street, Springfield, 62729");
```

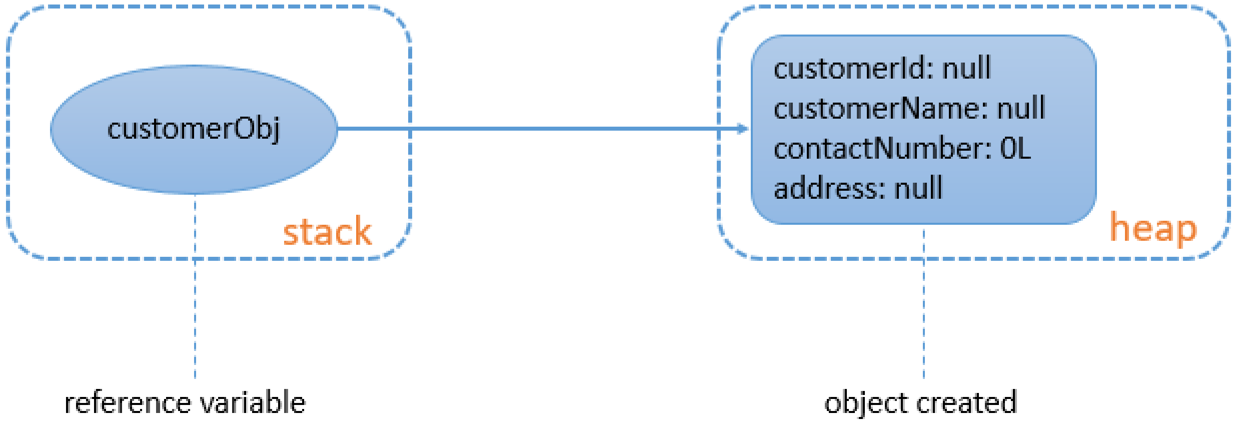
Memory Allocation

A new Customer object referenced by customerObj will be created in the memory.

But how does it happen?

The memory is logically divided into two primary sections - **Stack** and **Heap**.

- All local variables and method invocations are stored in the stack
- All objects along with their instance variables are stored in the heap

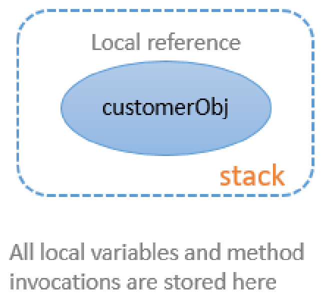


Please note that reference variables are also local variables. Reference variables are local variables which stores the address of another memory location.

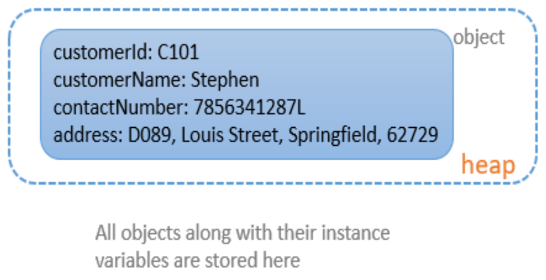
You will now have a look at memory management in detail.

Memory Allocation - Steps

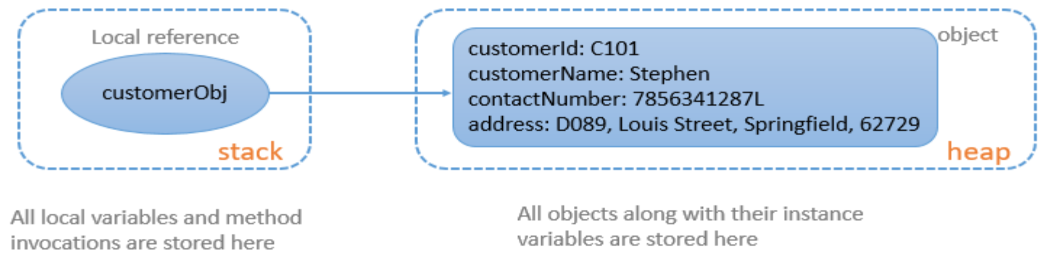
Step 1: The reference variable is created in the stack.



Step 2: The object is created in the heap.


customerId: C101
customerName: Stephen
contactNumber: 7856341287L
address: D089, Louis Street, Springfield, 62729
object
heap
All objects along with their instance variables are stored here

Step 3: The reference variable in the stack refers to the object in the heap.


Local reference
customerObj
stack
All local variables and method invocations are stored here
customerId: C101
customerName: Stephen
contactNumber: 7856341287L
address: D089, Louis Street, Springfield, 62729
object
heap
All objects along with their instance variables are stored here

Notes:

One reference variable can point to one and only one object at a time.

One object can be referenced by multiple reference variables at any given point of time.

Memory deallocation

Now that you have learnt how memory allocation happens, it is also important to know the process of deallocating the memory.

Sometimes, even though a resource in a program is unreachable or not in use, the memory used by that resource is not deallocated. This is called **Memory leak** and is undesirable.

In some languages, it is the programmer's responsibility for deallocating the memory occupied by such resources. Java, on the other hand, has a **garbage collector** which automatically deallocates the memory used by such resources. This prevents memory leak.

When an object does not have any reference, it becomes eligible for garbage collection.

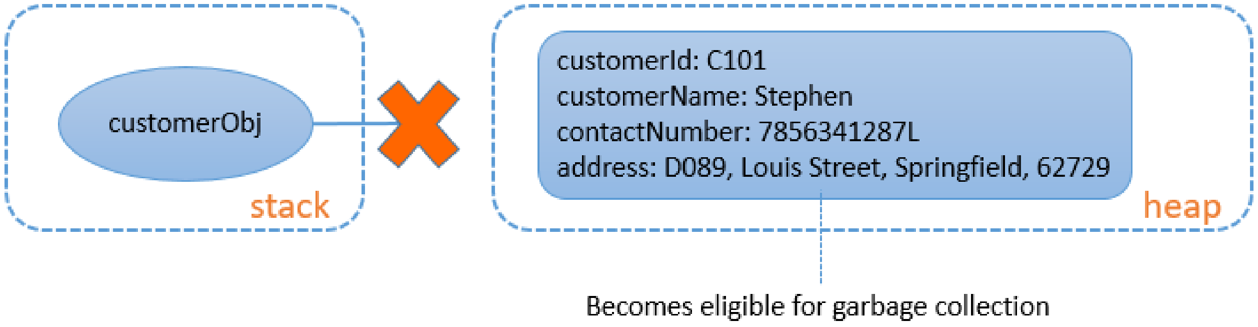
You might be wondering when does an object not have any reference and become eligible for garbage collection?

Let us look at some of the possibilities.

Garbage Collection – Case 1

Case 1 - Objects eligible for garbage collection

When the reference variable pointing to the object is initialized to null, the object will not have any reference.

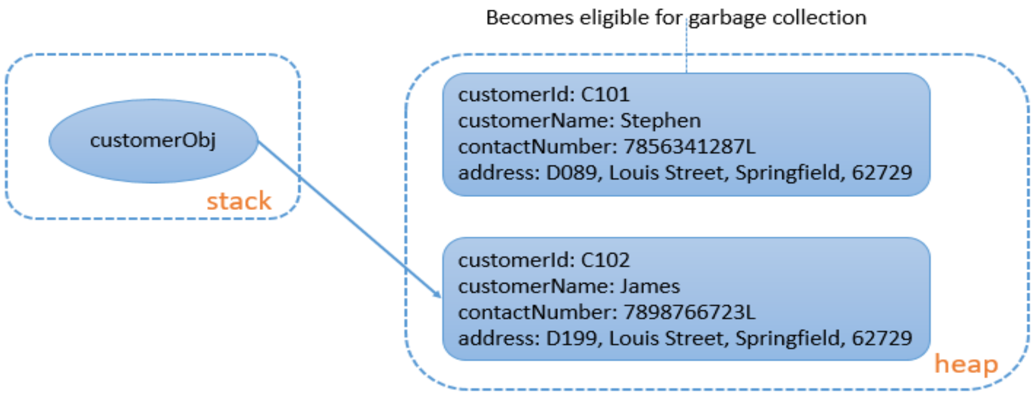

customerObj
stack
customerId: C101
customerName: Stephen
contactNumber: 7856341287L
address: D089, Louis Street, Springfield, 62729
heap
Becomes eligible for garbage collection

```java
public class TestMain {
   public static void main(String[] args) {
      //Object Creation
      Customer customerObj = new Customer("C101", “Stephen", 7856341287L,
       “D089, Louis Street, Springfield, 62729");
      //Reference variable initialized to null
      customerObj = null;
   }
}
```

Garbage Collection – Case 2

Case 2 - **Objects eligible for garbage collection**

When the reference variable is initialized to a new object and there is no reference to the previous object

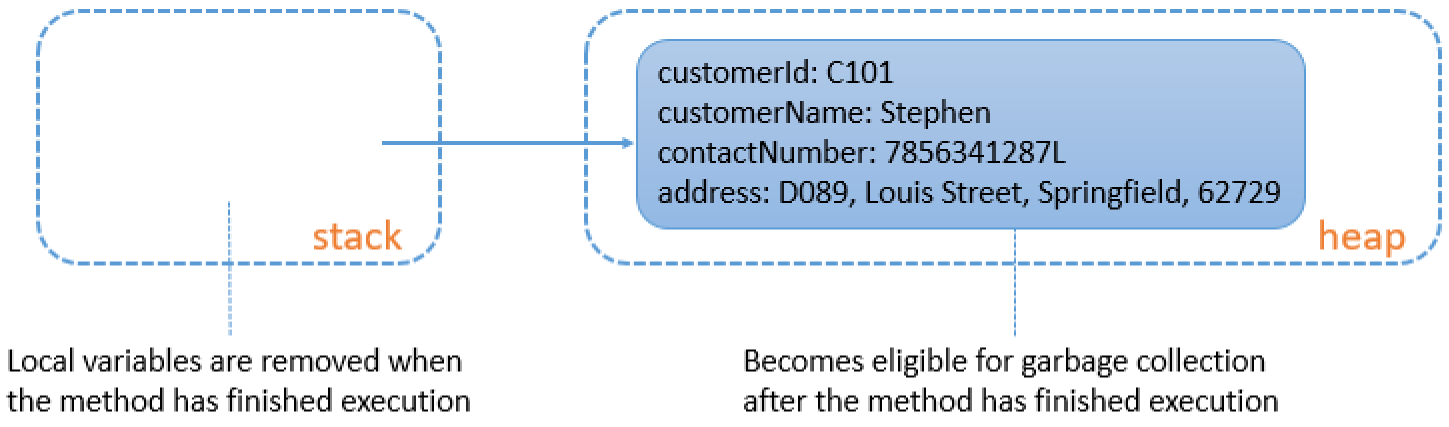

Becomes eligible for garbage collection
customerObj
stack
customerId: C101
customerName: Stephen
contactNumber: 7856341287L
address: D089, Louis Street, Springfield, 62729
customerId: C102
customerName: James
contactNumber: 7898766723L
address: D199, Louis Street, Springfield, 62729
heap

```java
public class TestMain {
    public static void main(String[] args) {
        //Object Creation
        Customer customerObj = new Customer("C101", "Stephen", 7856341287L,
         "D089, Louis Street, Springfield, 62729");
        //Reference variable initialized to null
        customerObj = new Customer("C102", "James", 7898766723L,
         "D199, Louis Street, Springfield, 62729");
    }
}
```

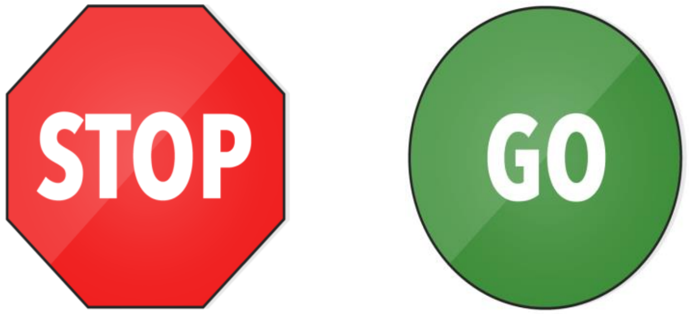
Garbage Collection – Case 3

Case 3 - **Objects eligible for garbage collection**

When a reference variable is local to some method, it will be removed from the stack as soon as the method finishes execution. The object pointed by the reference variable then becomes eligible for garbage collection.


customerId: C101
customerName: Stephen
contactNumber: 7856341287L
address: D089, Louis Street, Springfield, 62729
stack
heap
Local variables are removed when the method has finished execution
Becomes eligible for garbage collection after the method has finished execution

Time To Reflect



Trainees to reflect the following topics before proceeding.

- What is the Java Memory Management?
- Define Memory allocation
- Define Memory de-allocation
- When the Java Garbage collection is initiated?

Thank you

You have successfully completed

Java Memory Management

JAVA @11

Enumeration

Objective

After completing this session you will be able to understand,

- Java Enum
- Features of Enum
- Enum Implementation

enum – A grouped constant

Enumerations are group of named constants. All enums implicitly extend the java.lang.Enum class.

The enum fields are implicitly static and final, and hence are constant during compile time.

But they are instances of their enum type, constructed when the enum type is referenced for the first time.

Why we need this grouped constant?

Let us assume that a developer is creating online Pizza ordering application. He wants to allow the customers to choose the size of the pizza. The sizes allowed are small, medium and large only.

He realizes that having the type of the size variable as String has a chance of some developer entering any arbitrary size. And this could cause invalid processing. How can we stop the Pizza size to be initialized anything other than these three values.

Using enums allows us to limit the selection within a set of values.

Grouped constant

An **enum** is a datatype which contains a fixed set of constant values. For example,

directions (NORTH, EAST, WEST, SOUTH)

pizzaSize (SMALL, MEDIUM, LARGE)

Syntax:

```
public enum enum_name { constant1, constant2, ..., constant n }
```

Example:

```
public enum PizzaSize { SMALL, MEDIUM, LARGE }
```

This type can be declared to limit the usage to the above 3 values.

```
private PizzaSize size;
```

Let us see what all things can be done with enum based variables next.

Using enum

Let us look at some features about enums:

- Enums are considered as reference-types like classes and interfaces in Java, and hence, a programmer can define constructors, methods and variables, inside them.
- A static method called **values()** is automatically generated by the Java compiler for each enum. The **values()** method returns an array of all the constant values defined inside the enum.
- Enum variables can be used in an if statement or switch statement.

Enum – Types of statements

if - Statment

```
if(this.size.equals(PizzaSize.MEDIUM)){
        System.out.println("Size is Medium");
}
```

switch Statement

```
PizzaSize currentSize = PizzaSize.MEDIUM;
double discount = 0;
// using enum in switch case
switch (currentSize) {
case SMALL:
   discount = 10;
   break;
case MEDIUM:
   discount = 20.5;
   break;
case LARGE:
   discount = 30.2;
   break;
}
```

Enum – Types of statements

We can loop through the enum and print constant values.

for loop-

```
//values() method returns an array of all values inside enum
//ordinal() method can be used to display values assigned to enum constants
for (PizzaSize psize : PizzaSize.values()) {
        System.out.println(psize+" "+psize.ordinal());
}
```

Enumeration - Problem Statement

Pizza size is defined in Enum with SMALL, MEDIUM, LARGE sizes. Pizza size identified to get the respective discount.

This demo helps to learn different methods of Enumeration and to know the better usage.

```java
//Enumeration of Pizza Sizes
enum PizzaSize {
   SMALL, MEDIUM, LARGE
}
public class PizzaMain {
   private String pizzaName;
   private double price;
   //can take only three values SMALL, MEDIUM, LARGE.
   // Anyother value will give error
   private PizzaSize size;
   public PizzaMain(String pizzaName, double price, PizzaSize size) {
      this.pizzaName = pizzaName;
      this.price = price;
      this.size = size;
   }
   public double checkDiscount() {
      PizzaSize currentSize = this.size;
      double discount = 0;
      //enum can be used in switch case
      switch (currentSize) {
      case SMALL:
         discount = 10;
         break;
      case MEDIUM:
         discount = 20.5;
         break;
      case LARGE:
         discount = 30.2;
         break;
      }
      return discount;
   }
public static void main(String[] args) {
    PizzaMain myPizza = new PizzaMain("VegFeast", 500, PizzaSize.MEDIUM);
    System.out.println("Discount:" + myPizza.checkDiscount());
// displaying all Pizza sizes
//values() method returns an array of all values inside enum
//ordinal() method can be used to display values assigned to enum constants
     for (PizzaSize psize : PizzaSize.values()) {
       System.out.println(psize+" "+psize.ordinal());
     }
   }
}
```

Created By:	Kannan, Rajendran
Credential Information:	Trainer
Version and Date:	1.0, 21-Nov-2020

Enum – Exercise 1

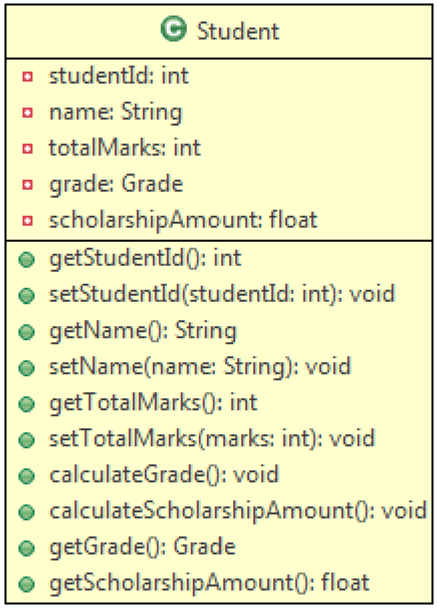
calculateGrade

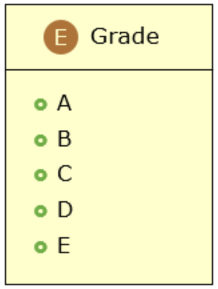
Total marks	Grade
>=250	A
>=200 & <250	B
>=175 & <200	C
>=150 & <175	D
else	E

calculateScholarshipAmount

Grade	Scholarship
A	5000
B	4000
C	3000
D	2000
E	Not applicable

A college has decided to provide scholarship to the students based on their grades and the grade will be decided based on their total marks. Create Student class and Grade enumeration according to the following diagrams.





Grade: (Enum)

Student: (Class)

Method Description:

calculateGrade(): (**To be implemented**)

This method computes the grade of a student based on his/her total marks, as per the following table.

calculateScholarshipAmount(): **(To be implemented)**

This method determines the scholarship amount depending on the grade:

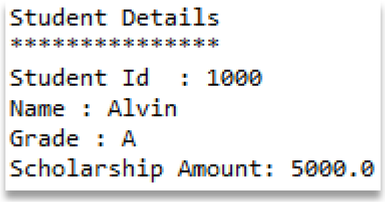
To test the above functionality, use a Tester class with the inputs mentioned below. Invoke appropriate methods and display the details as required.

Sample:

Input:

Attributes	Values
Student Id	1000
Student Name	Alvin
Student Total marks	280

Output:



```
Student Details
***************
Student Id  : 1000
Name : Alvin
Grade : A
Scholarship Amount: 5000.0
```

```java
//defining enum Grade
enum Grade{
   A,B,C,D,E;
}
class Student{
   private Integer studentId;
   private String name;
   private Integer totalMarks;
   private Grade grade;
   // Grade is the name of enum
   private float scholarshipAmount;
   // Generate Getter & Setter methods
   public void calculateGrade() {
      // write the logic to calculate the grade of the student
   }
   public void calculateScolarshipAmount() {
      // write the logic to calculate scholarship amount based on grade
   }
}
public class Scholarship {
   public static void main(String[] args) {
      Student student = new Student();
      // code here, to set the student values
      student.calculateGrade();
      student.calculateScolarshipAmount();
      System.out.println("Student Details");
      System.out.println("***************");
      // code here, to Print the Student Id, Name, Grade and Scholarship Amount
   }
}
```

Enum – Exercise 2

ShoppingCart is an electronics shop providing special discount to their customers. Once a customer selects their desired product, discount will be given based on product selection, if applicable.

Create **Shop** class and **Category** enumeration according to the following diagrams.

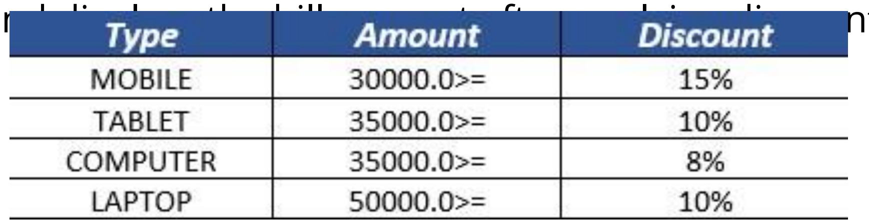
Method Description:

checkDiscountApplicability ():

This method checks whether the customer is applicable for discount or not. If applicable, it calls the addDiscount () method, else the appropriate message with bill amount is displayed.

addDiscount ():

This method calculates a [illegible] nt:



Type	Amount	Discount
MOBILE	30000.0>=	15%
TABLET	35000.0>=	10%
COMPUTER	35000.0>=	8%
LAPTOP	50000.0>=	10%

If above conditions are not satisfied then a general discount of 7% is applied.

```java
enum Category {
   MOBILE,TABLET,COMPUTER,LAPTOP
}
class Shop {
   private double amount;
   private Category type;
   private double discount;
   // Generate getters & setters method
   public void checkDiscountApplicability(double amount,Category type) {
      if(amount>=30000){
         System.out.println("congratulations you are applicable for discount");
         this.applyDiscount(type);
      } else {
         System.out.println("Thank you for shopping, your bill amount is: " + this.getAmount());
      }
   }
   public void applyDiscount(Category type) {
      //write your logic here
   }
}
public class ShoppingCart {
   public static void main(String[] args) {
      // Shop shop=new Shop(20000.0,Category.COMPUTER);
      // shop.checkDiscountApplicability(shop.getAmount(),shop.getType());
   }
}
```

Thank you

You have successfully completed

Java Enumeration

JAVA @11

Regular Expression

Objective

After completing this session you will be able to understand,

- Understand Regular Expression
- Define matches() and split() method
- Approach Regular Expression
- Grouping in Regular Expression

What is Regular Expression?

A Regular Expression or Regex is a special sequence of characters that help in matching or finding other strings or set of strings, using a specialized syntax. Regex are used in searching, editing and manipulating the text.

The Regex API (java.util.regex) provides the necessary classes and interfaces for working with regular expressions.

String class uses this API to support regex in its following methods: matches(), split(), replaceFirst(), and replaceAll().

Let us concentrate on the methods *matches()* and *split()* for now.

matches()

This method takes a regex as an argument and matches a given string with the specified regex pattern.

This method returns a Boolean value for the comparison done.

```java
package com.rntbci.learning.regex;
import java.util.regex.Pattern;

public class MatchesMain {
   public static void main(String[] args) {
   //Creating a Pattern through Regular Expression
      String regex = "Hello";
      //String to be compared with RegEx
      String input = "Hello";

      //Pattern Matching done using matches() method
      System.out.println(input.matches(regex));
      //Pattern Matching done using Pattern.matches() method
      System.out.println(Pattern.matches(regex, input));
      /**
       *  Output: true, true
       */
   }
}
```

It is interesting to note that, *input.matches(regex)* yields the same result as *Pattern.matches(regex, input)*.(Note: Pattern is one of the classes provided by the Regex API)

split()

This method splits a given string into an array of string on the basis of regular expression mentioned in the argument of the method.

```java
package com.rntbci.learning.regex;

public class SplitMain {
   public static void main(String[] args) {
      //Sample String
      String chainedString = "Lets-break-this-chain-and-be-free-!";
      //Creating Regex for splitting the String
      String regex = "-";
      //Splitting the String using split() method
      //split()divides the String and stores them in a Array of String
      String[] freeString = chainedString.split(regex);
 /**
  *  freeString = ("Lets", "break", "this", "chain", "and", "be", "free", "!")
  */
   }
}
```

How to write Regular Expression?

While working with strings, we must have wondered if there was an easier way to validate them.

What if we could specify a required pattern directly instead of checking things character by character?

What if we could provide a general pattern to search for instead of repeatedly looking for specific variations? And to make this possible, Java supports Regular Expressions.

Now the question arises: "How can we write such 'generalized' regular expressions?"

And before we go searching for the answers, let us see what a general regular expression looks like:

```
//Regular Expression for Name Validation:
String nameRegex = "([A-Za-z ]+)";

//Regular Expression for Email Id Validation:
String emailIdRegex = "([A-Za-z0-9-_]+)[@]([a-z]+)[.](com|in)";
```

How to write Regular Expression?

To write a generalized regular expression, we require the help of the following:

- **Literals**: They are the literal characters that have a literal meaning.

1. String literalRegex = "Hello";

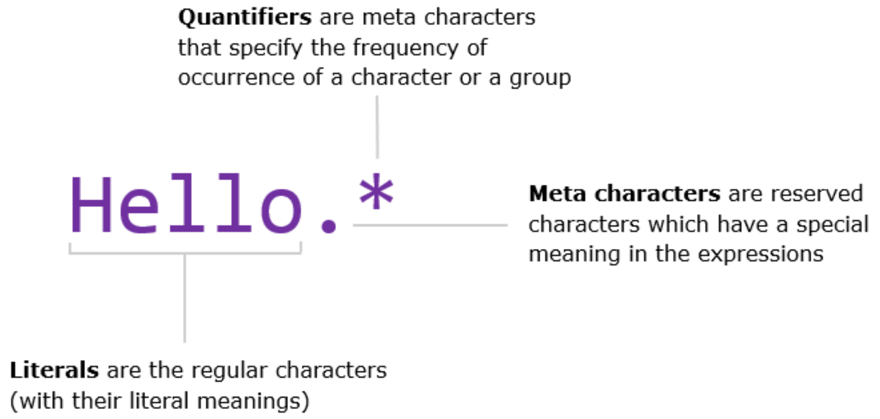
//The string must be exactly "Hello" to match this Regex pattern

- **Meta Characters**: They are reserved special characters/symbols which have special meaning in the Regex.

1. String metaRegex = "([A-Za-z0-9])“;
2. //The symbols and special characters have specific usages

- **Quantifiers**: They are a type of metacharacters that specify the frequency of occurence of a character or group.

1. String metaRegex = "([A-Za-z0-9]+)“; //The symbols "+" is a type of quantifier


Quantifiers are meta characters that specify the frequency of occurrence of a character or a group
Hello.*
Meta characters are reserved characters which have a special meaning in the expressions
Literals are the regular characters (with their literal meanings)

Meta Characters

Let's take a look at the different metacharacters.

Meta Char	Description	Example
[]	Bracket expressions create a character class to match a single character contained within the brackets. '_' can be used to specify a range.	[xyz] matches 'x', 'y' or 'z'. [a-z] matches any letter from 'a' to 'z'.
.	Matches any single character, except a newline. Inside a bracket expression, it becomes a literal dot.	b.t matches "bat", "bRt", "b8t", etc.
[^]	Matches a single character that is not within the brackets.	[^xyz] matches 'a', '6', etc.
I	"or" expression to match alternatives.	bat|cat matches "bat" or "cat"
()	Groups expressions to form sub-expressions. Also used to capture groups.	Ma(nn|tt)er matches "Matter" or "Manner"
\n	Matches the nth captured sub-expression group. Groups are numbered from left to right.	([a-z])\1 matches "dd", "hh", etc.

Note:

If meta characters are needed to be used as literals, they have to be escaped with a double backslash.

To refer to captured groups in the replacement methods, $<group no.> is used. E.g. str.replaceAll(regex, "$1");

Next, let us look at quantifiers.

Quantifiers

Let's take a look at the different quantifiers.

Meta Char	Description	Example
?	Matches the preceding element zero or one time.	Ba?it matches "Bait" and "Bit"
*	Matches the preceding element zero or more times.	10*1 matches "11", "1001", etc.
+	Matches the preceding element one or more times.	10+1 matches "101", "1001", etc.
{m}	Matches the preceding element exactly m times.	10{4}1 matches "100001"
{m,}	Matches the preceding element m or more times.	10{3,}1 matches "10001", "100001", "1000000001", etc.
{m,n}	Matches the preceding element minimum m and maximum n times.	xy{2,3}z matches "xyyz" and "xyyyz"

Predefined Character classes

Now that we have seen different metacharacters and quantifiers, let's see some predefined classes used in Regex.

These "predefined classes" can be used as alternatives to standard Regex written using only meta characters and literals.

Meta Char	Description	Example
\w	Alphanumeric characters and the underscore	[A-Za-z0-9_]
IW	Non-word characters	[^A-Za-z0-9_]
\d	Digits	[0-9]
\D	Non-digits	[^0-9]
\s	Whitespace characters	[\t\n\f\r)
\S	Non-whitespace characters	[^\t\n\f\r]

Note:

"\w" is a meta character. But in Java, '\' is an escape character. Hence, we have to escape it using another backslash. E.g. "[\\w]+"

Now, let us see how you can write a regular expression.

Approaching Regular Expression

Let's now see how to approach a pattern matching problem using the Regular Expression method.

But before we proceed, it must be noted and understood that there can be multiple solutions to a problem.

Let's consider a situation where we have to validate a Courier Consignment Tracking Number. The required format/pattern for the tracking number is given as:

Format of Courier Consignment Tracking Number:

<<2_Uppercase_Characters>>-<<9_digits>>:<<2_Uppercase_Characters>>

Example: "EK-860619645:IN"

Let us observe all the character sets the string is containing.

This pattern has only Uppercase Alphabets, Digits and Two Special Characters '-' and ':'. Next, let us find the frequency of occurrences for the character classes.

With that, let's break the pattern into smaller sub-patterns:

Approaching Regular Expression

The first sub-pattern is having two uppercase characters followed by "-" character. So, the Regex for this sub-pattern can be written as:

```
String firstSubRegex = "[A-Z]{2}[-]";
```

The second sub-pattern has 9 digits followed by ":" character. So, the Regex for this sub-pattern can be written as

```
//First Alternative:
String secondSubRegex = "[0-9]{9}[:]";
//Another Alternative:
String secondSubRegex = "[\\d]{9}[:]";
```

The last sub-pattern is having only two uppercase characters again. So, the Regex for this sub-pattern can be written as:

```
String thirdSubRegex = "[A-Z]{9}“
```

Hence, the overall matching pattern can be written as:

```
//Concatenation of all three Sub-Regex:
String finalRegexPatternConcat = firstSubRegex + secondSubRegex + thirdSubRegex;
//Alternatively the single Regex:
String finalRegexPatternComplete = "([A-Z]{2}[-])([\\d]{9}[:])([A-Z]{2})";
```

Examples on Regular Expression

Now let us conclude Regular Expressions by looking into following situational examples:

1. Requirement: To search the pattern "App" in the given string "Application".

```
String searchStr = "Application";
String regexStr = "App.*";
System.out.println(searchStr.matches(regexStr));                //Output: true
```

2. Requirement: To search the pattern having two characters in between A and l in the given string "A%(lication".//First Alternative:

```
String searchStr = "A%(lication";
String regexStr = "A..lication";
System.out.println(searchStr.matches(regexStr));                //Output: true
```

3. Requirement: To search for a digit between M and t in the given string "M4thematics".

```
String searchStr = "M4thematics";
String regexStr = "M\\dt.*";
System.out.println(searchStr.matches(regexStr));                //Output: true
```

4. Requirement: To search for a number between 4 and 8 in between X and Y in the given string.

```
String searchStr = "X9Y";
String regexStr = "X[4-8]Y";
System.out.println(searchStr.matches(regexStr));                //Output: false
```

Examples on Regular Expression

5. Requirement: To search for the pattern "Hell" or "Fell" in the given string "Fellow".

```
String searchStr = "Fellow";
String regexStr = "(Hell|Fell).*";
System.out.println(searchStr.matches(regexStr));            //Output: true
```

6. Requirement: To check for the space after "Air" in the given string "Air line".

```
String searchStr = "Air line";
String regexStr = "Air\\s.*";
System.out.println(searchStr.matches(regexStr));            //Output: true
```

7. Requirement: To check if a number is found 0 or n times after X in the given string.

```
String searchStr = "X4756Y";
String regexStr = "X\\d*Y";
System.out.println(searchStr.matches(regexStr));            //Output: true
```

8. Requirement: To check if a number is found 1 or n times after M in the given string.

```
String searchStr = "M4N";
String regexStr = "M\\d+N";
System.out.println(searchStr.matches(regexStr));            //Output: true
```

Examples on Regular Expression

9. Requirement: To check if a number is found 0 or 1 times after A in the given string.

```
1.    String searchStr = "M2N";
2.    String regexStr = "M\\d?N";
3.    System.out.println(searchStr.matches(regexStr));                    //Output: true
```

10. Requirement: To check if 3 digits are present after A in the given string.

```
1.    String searchStr = "M42N";
2.    String regexStr = "M\\d{3}N";
3.    System.out.println(searchStr.matches(regexStr));                    //Output: false
```

Grouping in Regular Expression

Grouping of a Regular Expression is done to divide a complete pattern into smaller groups of the pattern. This is done by using the parenthesis "()" brackets. Let us look at the various situations where grouping is necessary:

1. Grouping is necessary to group a combination of letters or words together:

```
//Regex to search between two alternatives: "Scanner" or "Scammer":
String regex = "Sca(nn|mm)er";
String demoStr = "Scammer";
System.out.println(demoStr.matches(regex));
```

```
//Regex to search for a particular word in a String
String regex = ".*(dog).*";
String demoStr = "Bowser was my dog."
System.out.println(demoStr.matches(regex));
```

2. Grouping is necessary to divide a bigger pattern into smaller patterns. Let us take a previous example to observe this better.

We had the format for Courier Consignment Tracking Number as "*<<2_Uppercase_Characters>>-<<9_digits>>:<<2_Uppercase_Characters>>*". And this pattern could be divided into the following three parts:

- Regex Sub-Pattern 1: The first sub-pattern is having two uppercase characters followed by "-" character - "[A-Z]{2}[-]"
- Regex Sub-Pattern 2: The second sub-pattern has 9 digits followed by ":" character - "[\\d]{9}[:]"
- Regex Sub-Pattern 3: The last sub-pattern is having only two uppercase characters again - "[A-Z]{2}

```
//Grouping of Regex done according to Sub-patterns using "()":
String regex = "([A-Z]{2}[-])([\\d]{9}[:])([A-Z]{2})";
```

Grouping in Regular Expression

Grouping is necessary for validating repeating/non-repeating sequences. In order to do so, we also use Back Referencing along with Grouping:

```
//Regex to search for repeating sequence in the String:
String passCodeRegex = ".*(\\d)\\1+.*";
String passCodeStr = "2455254";
System.out.println(passCodeStr.matches(passCodeRegex));
```

Here, "\\1" after "(\\d)" in the regular expression is called a backreference. It helps in checking the group mentioned in the regex is repeating or not.

```
//Regex having three groups being Back-Referenced
//The three groups must repeat after the ":" symbol in the String to match the pattern.
String repeatRegex = "([A-Za-z0-9]+) (\\d+) ([A-Z]+) [:] \\1 \\2 \\3";
String repeatStr = "Tom123 9090 JERRY : Tom123 9090 JERRY";
```

In the above example, "\\1" refers to the first group "([A-Za-z0-9])", "\\2" refers to "(\\d+)" and "\\3" refers to "([A-Z]+)".

And so, the regex expects the string to have all the three groups repeat itself whenever the back referencing is done.

RegEx – Exercise 1

The regex API (java.util.regex) provides classes and interfaces to work with regular expressions.

The String class uses this API to support regex method matches().

```java
package com.rntbci.learning.regex;

public class RegexDemoOne {
    public static void main(String[] args) {
        String str = new String("We are going to learn Regular Expression in Java");
        //Declaring Regular Expressions For Comparisons
        String regex1 = "[A-Z].*";
        String regex2 = ".*to.*";
        /**
         *      Different Uses of matches() method:
         *      Note: matches() returns Boolean value (True/False) as result.
         */
        //1. Checking the given String starts with an Uppercase Letter:
        System.out.println("Does the string start with an Uppercase Letter? : "+str.matches(regex1));

        //2. Checking the given String contains a particular Substring in it:
        System.out.println("Does the string contain the word 'to'? : " +str.matches(regex2));
        String str2 = "Thomas34";
        //Declaring the pattern as a Regular Expression:
        /**
         *      The Required Pattern is:
         *          1. First letter of String should be Uppercase [A-Z]
         *          2. Atleast one letter in lowercase ([a-z]+)
         *          3. should end with any two digit number \\d{2}
         */
         String regex3="[A-Z]([a-z]+)\\d{2}";
         //3. Checking the given String follows a given Pattern:
         System.out.println("Does ("+str2+") match with required pattern ? : "+str2.matches(regex3));
    }
}
```

RegEx – Exercise 2

Leeroy wants to validate the personal details which should be in the following format:

- Name: It can only contain characters and spaces.
- Email Id: <firstname>.<lastname>@<any>.<com/in>
- User Id: <firstname>#<any 2 digits><lastname>
- Account Serial Id: WoW-<any 7 alphabetic code><any 5 digits><Same 7 alphabetic code>_<any 5 alphanumeric code>

On successful execution of code it will return the result as "true" or "false", let's see how we can achieve this using Regex.

```java
package com.rntbci.learning.regex;

public class PersonalDetailValidatorDemo {
    public static void main(String[] args) {
        //Personal Details:
        String name = "Leeroy Jenkins";
        String emailId = "leeroy.jenkins@wow.com";
        String userId = "leeroy#78jenkins";
        String accSerialId = "WoW-ABilOpZ00523ABilOpZ_a00Z9";

        //Splitting Full Name Into an Array:
        String[] nameSplit = name.toLowerCase().split(" ");

        /**
         * Regular Expression for each field according to the requirements:
         */
        //Regular Expression for Name Validation:
        String nameRegex = "([A-Za-z ]+)";
        //Regular Expression for Email Id Validation:
        String emailIdRegex = nameSplit[0]+"."+nameSplit[1]+"[@]([a-z]+)[.](com|in)";
        //Regular Expression for User Id Validation:
        String userIdRegex = nameSplit[0]+"[#]([0-9]{2})"+nameSplit[1];
        //Regular Expression for Account Serial Id Validation:
        String accSerialRegex = "(WoW)[-]([A-Za-z]{7})(\\d{5})\\2[_]([A-Za-z0-9]{5})";

        /**
         * Matching the Personal Details with the pattern specified:
         */
        //Validating Name:
        System.out.println("Does the name ("+name+") match the pattern? : "+name.matches(nameRegex));
        //Validating Email Id:
        System.out.println("Does the email ID ("+emailId+") match the pattern? : "+emailId.matches(emailIdRegex));
        //Validating User Id:
        System.out.println("Does the user ID ("+userId+") match the pattern? : "+userId.matches(userIdRegex));
        //Validating Account Serial Id:
        System.out.println("Does the account serial ID ("+accSerialId+") match the pattern? : "+accSerialId.matches(accSerialRegex));
    }
}
```

Thank you

You have successfully completed

Java Regular Expression

JAVA @11

Unit Testing

Objective

After completing this session you will be able to understand,

- Unit Testing
- Introduction to Junit
- Configuring Junit
- Test cases and Test Methods

Unit Testing

You have been writing code but it is very important to deliver a bug free code.

Testing is the process of checking the functionality of an application to ensure that it runs as per requirements and does not have a bug.

It is the responsibility of the developer to test the module or method that he/she has developed before the code is integrated with others code.

Testing of single small units of code such as a method or a class is called **unit testing**.

So, it is the responsibility of the developers to deliver a **unit tested code**. It plays a critical role in delivering quality products to customers.

Unit testing can be done manually or can be automated using tools.

Unit Testing – Manual vs Automated

Manual Testing	Automated Testing
Executing test cases manually without any tool	Executing test cases using a tool
Time consuming and tedious - since test cases are executed manually, it is slow and tedious	Fast - significantly faster than manual testing
Huge investment in human resources - test cases need to be executed manually and therefore more testers are required.	Less investment in human resources - test cases are executed using tools, so less testers are required.
Less reliable - as it has to account for human errors	More reliable - precise and reliable

Unit Test Case

Unit testing is performed using **unit test cases**.

A unit test case is a part of code, which ensures that another part of code (method) works as expected.

A unit test case is characterized by a known input and an expected output, which is worked out before the test is executed.

There must be at least two unit test cases for each requirement – one positive test and one negative test. If a requirement has sub-requirements, each sub-requirement must have at least two test cases as positive and negative.

To achieve the desired results quickly, a test framework is required. There are several frameworks available for automating unit tests. **JUnit** and TestNG are among the popular ones for Java.

JUnit based unit testing will be discussed in detail later in this course.

Testing Introduction

Testing is an essential part while developing an application or even a small code.

Testing ensures that code is functioning properly for all possible cases.

Manual testing of code is an error prone process and time consuming, especially when it comes to larger applications with increased code size.

Thus, performing automated testing is more reliable and saves lot of time.

JUnit

There are several frameworks available for automating unit tests. **JUnit** and **TestNG** are among the popular ones for Java.

In this course, you will be using JUnit.

JUnit is an open-source unit testing framework for Java programming language. It provides classes to write and run automated tests.

Features of JUnit

- open source framework used for writing and running tests
- less complex and takes less time
- provides immediate feedback
- shows test progress

Configuring JUnit

We need JARs for performing JUnit testing.

Java Archive (JAR) is a file containing packaged class files and associated resources. JARs are usually used to distribute applications or libraries. Projects which needs the content of an external JAR need to have the JAR included in their build path.

The following JARs are required for JUnit testing.

junit-4.12.jar

hamcrest-core-1.3.jar

You can download the JARs from here.

You will now see how to add JARs to a project.

Adding JARs

To add the JARs required for the project, follow the steps given below.

Step 1: Right click on the core-app project. Choose **Build Path** -> **Configure Build Path.**

Step 2: In the 'Properties' window, select 'Libraries' tab as shown.

Step 3: Click on Add External JARs. to add new JARs.

Step 4: Browse and navigate to the path where you have downloaded the required JARs. Select both the JARs and click on 'Open'.

Step 5: New JARs will be included in the Libraries tab. Click OK.

JUnit – Test cases and Test methods

Now that you have added the required JARs, you will now see how to implement testing using JUnit.

After placing an order in core-app, bill details have to be generated.

Payment mode needs to be validated before generating the bill as the restaurant allows only five payment options - Credit Card, Debit Card, PayPal, Amazon Pay and Google Pay.

You need to write automated test cases using JUnit for validating the payment mode.

```java
public class Bill {
   private String paymentMode;
   // other variables and methods
   public Bill(String paymentMode) {
      this.paymentMode = paymentMode;
   }
   public boolean validatePaymentOption() {
      if (paymentMode.equals("Credit Card")
          || paymentMode.equals("Debit Card")
          || paymentMode.equals("PayPal")
          || paymentMode.equals("Amazon Pay")
          || paymentMode.equals("Google Pay")) {
          return true;
      }
      return false;
   }
}
```

JUnit - Assertion

Working with JUnit starts with creation of test methods.

A test method can either make assertions or expect exceptions.

Assertions are statements which compare the **expected results** with **actual results** returned from the method.

If they match, the test case passes else the test case fails.

- **org.junit.Assert** class is used for making assertions
- The org.junit.Assert class provides different methods to compare the expected result and actual result
- Any method that is written with assertions cannot be classified as a test method. To classify a method as a test method, you need to use the annotation **org.junit.Test**.
- Once a class has a test method, it is called a test class or a test case.

Consider the test case given below.

```java
public class BillTest {
   @Test
   public void validatePaymentOptionTestValid() {
      Bill bill = new Bill("PayPal");
      Assert.assertTrue(bill.validatePaymentOption());
   }
}
```

The above test method **validatePaymentOptionTestValid()** tests a condition for the **validatePaymentOption()** of the Bill class. If the method has been properly implemented, then the method will return true and the test will pass.

Assert Class

The **Assert** class provides static methods for testing a variety of conditions. These methods accept actual and expected results for comparison and determine whether a test method will pass or fail.

When a test fails, the test method throws an **AssertionException**. Optional error messages can be specified for Assertion exceptions. These exceptions are caught by JUnit and their messages are displayed.

Here are some useful methods of the Assert class:

Method	Description
void assertEquals(expected, actual)	fails if expected result is not equal to the actual result
void assertSame(expected, actual)	fails if expected object and actual objects are not referring to the same object
void assertTrue(boolean test)	fails if the test condition results in false
void assertFalse(boolean test)	fails if the test condition results in true
void assertNull(Object object)	fails if the object is not null
void assertNotNull(Object object)	fails if the object is null

JUnit Test case Execution

Test cases (test classes) can be run as a JUnit test as shown below.

Right click on the file and select 'Run As'->'Junit Test’.

The JUnit **Runner** is responsible for constructing the instances of the test classes before running the tests. It is also responsible for making them available for garbage collection after the tests.

JUnit Report

The image given below shows the JUnit report. It provides:

number of test methods executed

number of test methods having errors

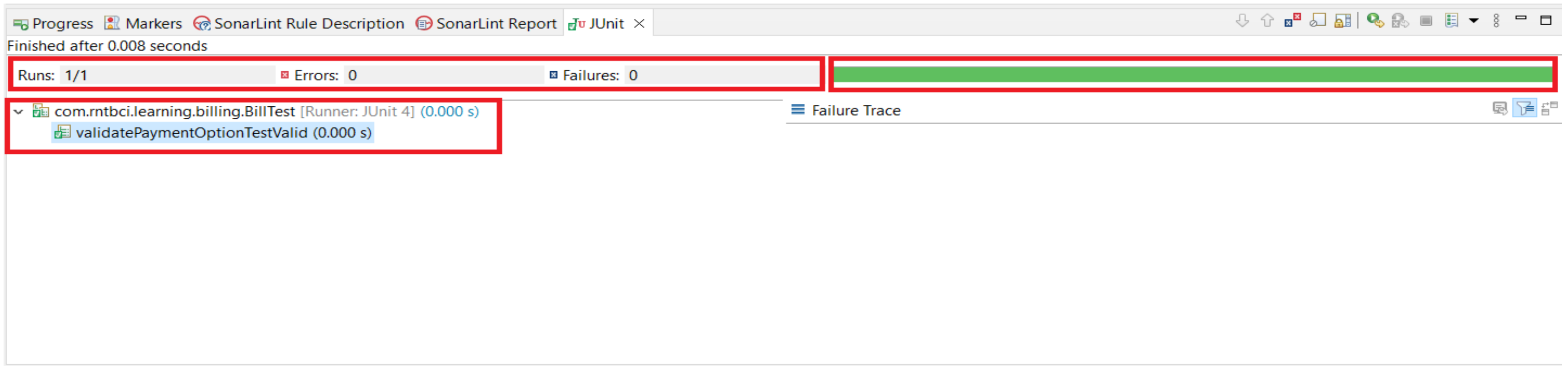
number of failures among the test methods

list of all the test methods

failure trace, in case of any failure

final test status with color coding

- green, if all the test cases pass
- red, if any test case fails



Finished after 0.008 seconds

Runs: 1/1 Errors: 0 Failures: 0

com.rntbci.learning.billing.BillTest [Runner: JUnit 4] (0.000 s)
validatePaymentOptionTestValid (0.000 s)

Testing Exceptional Flow

Consider the same scenario where the payment option is validated. In this case, an exception is being thrown for invalid payment modes.

You have already seen how to write automated test cases and use assert statements.

But for the above scenario, you can’t use assert statements for testing as the method is throwing an exception.

Here, an exception with proper message is expected for invalid payment mode.

```java
public boolean validatePaymentOption()throws Exception {
    if (paymentMode.equals("Credit Card")
         || paymentMode.equals("Debit Card")
         || paymentMode.equals("PayPal")
         || paymentMode.equals("Amazon Pay")
         || paymentMode.equals("Google Pay")) {
         return true;
    }
    return false;
}
```

Expecting Exception

JUnit allows us to create rules for adding a new behavior or redefining the behavior of each test method in a test class.

The **@Rule** annotation is used for this purpose. It marks the public fields with the type **TestRule** which is an abstract class, and allows developers to create custom rules.

TestRule has a lot of implementation classes, out of which **ExpectedException** is one which can be used to test for exception types and messages.

It has the following useful methods:

Method	Description
ExpectedException none()	returns an object of ExpectedException that expects no exceptions, i.e., an empty rule
void expect(Throwable t)	verifies that a code throws a specified exception
void expectMessage(String substring)	verifies that a code throws an exception with a message containing the specified text.

Expecting Exception

The Test method for the validatePaymentOption() to test the exception is as follows:

```java
public class BillTest {
   @Rule
   public ExpectedException ee = ExpectedException.none(); // creates an empty rule

   @Test
   public void validatePaymentOptionTestInValid()throws Exception {
      ee.expect(Exception.class); // expecting the type of exception
      // expecting a text in the exception
      ee.expectMessage("Please provide valid payment option.");
      // creating a bill with invalid payment option
      Bill bill = new Bill("Paypal");
      // invoking the method to validate
      bill.validatePaymentOption();
   }
}
```

Once the rule has been created, it can be modified to expect a certain type of exception (using **expect()**), or an exception with a certain text (using **expectMessage()**), or both.

The test case will pass if an exception of the specified type and containing the specified text is thrown.

Note: The expect() and expectMessage() methods can be used individually as per the requirement.

TestSuites

An enterprise grade application will have huge number of test cases.

Is it feasible to run all the test cases separately?

Obviously NOT!!!

It would be much better if we can group multiple related test cases and execute them together.

JUnit provides **test suits** to create groups of test classes so that the code maintainability is improved.

The following annotations are used for creating test suites:

Method	Description
@RunWith(Suite.class)	a class with this annotation becomes a test suite class and is invoked by JUnit to run the test classes, instead of the built in runner.
@Suite.SuiteClasses	specifies the test classes to be grouped together and executed when their test suite class runs

TestSuites

Example:

```
@RunWith(Suite.class)
@SuiteClasses({ BillTest.class, BillExceptionTest.class})
public class TestSuiteDemo {

}
```

The above class when executed using JUnit will execute all the test methods present in the classes mentioned in @SuiteClasses

If BillTest.class and BillExceptionTest.class contain two test methods each, then, the test suite given here executes four different test methods and provides the report.

Code Coverage

While testing, it is also important to ensure that each and every part of the code gets tested. This can be ensured using **Code coverage.**

Code coverage aims at determining the extent to which code is tested during unit testing.

With code coverage, you can determine the parts of the code which were not executed by test cases. You can then modify your unit tests to ensure those parts of the code are executed.

The larger the code coverage, better is the chance of having a bug-free code.

How do you measure code coverage?

You can make use of coverage tools to measure code coverage. Some commonly used coverage tools are as follows:

- EclEmma
- JCov
- JaCoCo
- Cobertura
- Emma

We will be using **EclEmma** in this course.

EclEmma

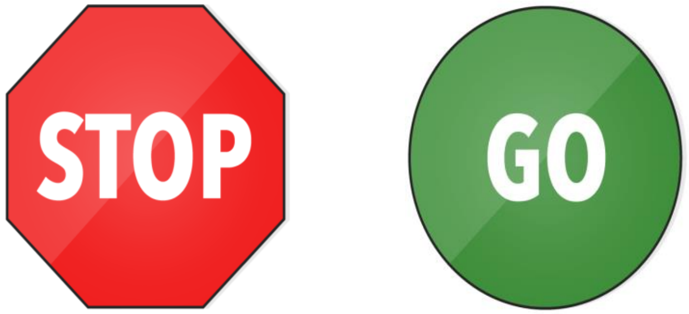
EclEmma is a free Java code coverage tool for Eclipse. It brings code coverage analysis directly into the Eclipse workbench.

Some of the features of EclEmma are:

- JUnit test runs can directly be analyzed for code coverage
- Coverage results are immediately summarized and highlighted in the Java source code editors
- EclEmma does not require modifying your projects or performing any other setup

You will now see how to find out code coverage using EclEmma.

Time To Reflect



Trainees to reflect the following topics before proceeding.

- What is the class you use for processing dates?
- How to create customized format?
- What class can we use for formatting dates based on locales?

Thank you

You have successfully completed

Java Unit Testing