# Strictfp

* strictfp ensures to give the same results in all platforms while performing operations on floating-point numbers
* it is a modifier and it can be used with either class, interface, or methods only.
* if the class of interface is prefixed with strictfp keyword then all methods of class and interface are implicitly strictfp methods only.

|  |
| --- |
| **package** miscellaneous;  **strictfp** **public** **class** Strictfp  {  **public** **static** **void** main(String[] args)  {  **float** a = 10.0f;  **float** b = 3.0f;  System.***out***.println(a/b);  }  }  //OUTPUT  3.3333333 |

# Assertions

* An assertion is a condition that must be true during program execution.
* Assertions are used to identify logical errors
* By default, assertions are disabled. We need to enable to use them.

|  |
| --- |
| **package** miscellaneous;  **import** java.util.Scanner;  **public** **class** AssertionsExample  {  **public** **static** **void** main(String[] args)  {  Scanner s = **new** Scanner(System.***in***);  System.***out***.println("Enter any number between 1 and 10 - ");  **int** x = s.nextInt();  **assert**((x>0) && (x<=10));  System.***out***.println(x);  }  }  //OUTPUT  Enter any number between 1 and 10 -  10  10  Enter any number between 1 and 10 -  100  Exception in thread "main" java.lang.AssertionError  at miscellaneous.AssertionsExample.main(AssertionsExample.java:12) |

# Varargs

* It means variable arguments.
* This feature allows to pass 0 to any number of arguments to a method.
* Varargs must be the last argument otherwise a compile-time error occurs

Void display(int a, int b, int…. c) { } => Valid

Void display(int… a, int b, int c) { } => Invalid, Compile-time error.

|  |
| --- |
| **package** miscellaneous;  **public** **class** VarargsExample  {  **public** **static** **void** main(String[] args)  {  Demo d = **new** Demo();  d.add();  d.add(10);  d.add(10,20);  d.add(10,20,30);  }  }  **class** Demo  {  **public** **void** add(**int**... a)  {  System.***out***.println("Addtions");  }  } |

# Enumerations

* This feature allows us to create new data type.
* All enumeration literals are implicitly static.
* Java compiler generates .class file for enumerations also.

|  |
| --- |
| **package** miscellaneous;  **enum** Day  {  ***Mon***,***Tue***,***Web***,***Thu***,***Fri***,***Sat***,***Sun***  };  **public** **class** EnumerationsExample  {  **public** **static** **void** main(String[] args)  {  Day d = Day.***Tue***;  System.***out***.println(d);  }  }  OUTPUT  Tue |

# Annotations

* Annotations are meta tags that are used to pass some additional information to compiler.
* In the below example, @Override annotation informs compiler that show() method is an overriding method.

|  |
| --- |
| **package** miscellaneous;  **public** **class** AnnotationExample **extends** xyz  {  **public** **static** **void** main(String[] args)  {  AnnotationExample obj = **new** AnnotationExample();  obj.show();    }  @Override  **void** show()  {  System.***out***.println("Show method in main class");  }  }  **class** xyz  {  **void** show()  {  System.***out***.println("Show method in demo class");  }  }  //OUTPUT  Show method in main class |

# Binary Literals

* Binary Literals always starts with 0b.

|  |
| --- |
| **package** miscellaneous;  **public** **class** BinaryLiteralsExample  {  **public** **static** **void** main(String[] args)  {  **int** a = 0b1000;  System.***out***.println(a);  }  }  //OUTPUT  8 |

# Underscores in numeric literals

* This feature allows to write any number of underscore symbols in integer literals and floating-point literals.
* Whenever we want to give a break, we will include one underscore.
* We cannot give underscore in beginning and end.
* Output will not include underscores.
* This is only for reading purpose.

Examples.

Int a = \_55 => Error

Int b = 5\_5 => Valid

Int b = 55\_ => Error

Float a = 5.3\_4f => valid

Float a = 5\_.34f => error

Float a = 5.\_34f => error

Float a = 5.34\_f =>error

Float a = 5.34f\_ => error

|  |
| --- |
| **package** miscellaneous;  **public** **class** UnderscoreExample {  **public** **static** **void** main(String[] args)  {  **long** phoneno = 425\_350\_8630l;  System.***out***.println(phoneno);  }  }  //OUTPUT  4253508630 |

# Strings in Switch statement

* This feature allows to pass strings in switch statement.

|  |
| --- |
| **package** miscellaneous;  **import** java.io.BufferedReader;  **import** java.io.IOException;  **import** java.io.InputStreamReader;  **import** java.util.Scanner;  **public** **class** StringsInSwitchStatementExample {  **public** **static** **void** main(String[] args) **throws** IOException  {  Scanner s = **new** Scanner(**new** InputStreamReader(System.***in***));  System.***out***.println("Enter - ");  String str = s.next();  **switch** (str)  {  **case** "Gowtham":  {  System.***out***.println("Father.");  **break**;  }  **case** "Anusha":  {  System.***out***.println("Mother.");  **break**;  }  **case** "Siddhu":  {  System.***out***.println("SON.");  **break**;  }  **default**:  System.***out***.println("Others.");  }  }  } |

# New Date and Time API

Java.time package

Classes

LocalDate

LocalTime

LocalDateTime

|  |
| --- |
| **package** miscellaneous;  **import** java.time.LocalDate;  **public** **class** DateTimeExample {  **public** **static** **void** main(String[] args)  {  LocalDate date = LocalDate.*now*();  System.***out***.println(date.*now*());  System.***out***.println(date.plusYears(10));  System.***out***.println(date.plusMonths(1));  System.***out***.println(date.plusWeeks(1));  System.***out***.println(date.plusDays(1));  System.***out***.println(date.minusYears(10));  System.***out***.println(date.minusMonths(1));  System.***out***.println(date.minusWeeks(1));  System.***out***.println(date.minusDays(1));  }  }  //OUTPUT  2022-05-28  2032-05-28  2022-06-28  2022-06-04  2022-05-29  2012-05-28  2022-04-28  2022-05-21  2022-05-27 |