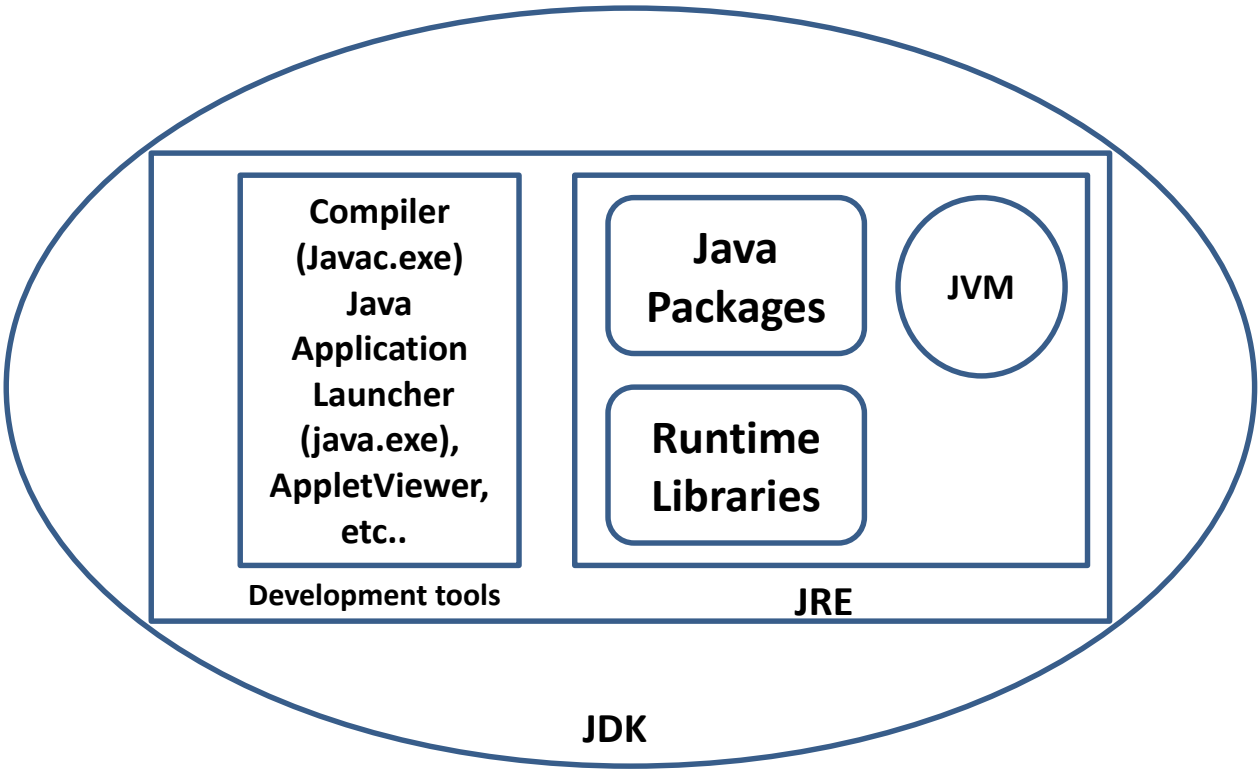
History of JAVA.

- Java was initially developed in 1991 named as “oak” but was renamed “Java” in 1995.
- Originally designed for small, embedded systems in electronic appliances like set-top boxes.
- The primary motivation was the need for a platform-independent language that could be used to create software to be embedded in various consumer electronic devices.
- Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).
- It promised Write Once, Run Anywhere (WORA), providing no-cost run-times on popular platforms.
- Java 2, new versions had multiple configurations built for different types of platforms. J2EE included technologies and APIs for enterprise applications typically run in server environments, while J2ME featured APIs optimized for mobile applications.
- The desktop version was renamed J2SE. In 2006, for marketing purposes, Sun renamed new J2 versions as Java EE, Java ME, and Java SE, respectively.
- On 13 November 2006, Sun released much of Java as free and open-source software (FOSS), under the terms of the GNU General Public License (GPL).
- On 8 May 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

What is Java?

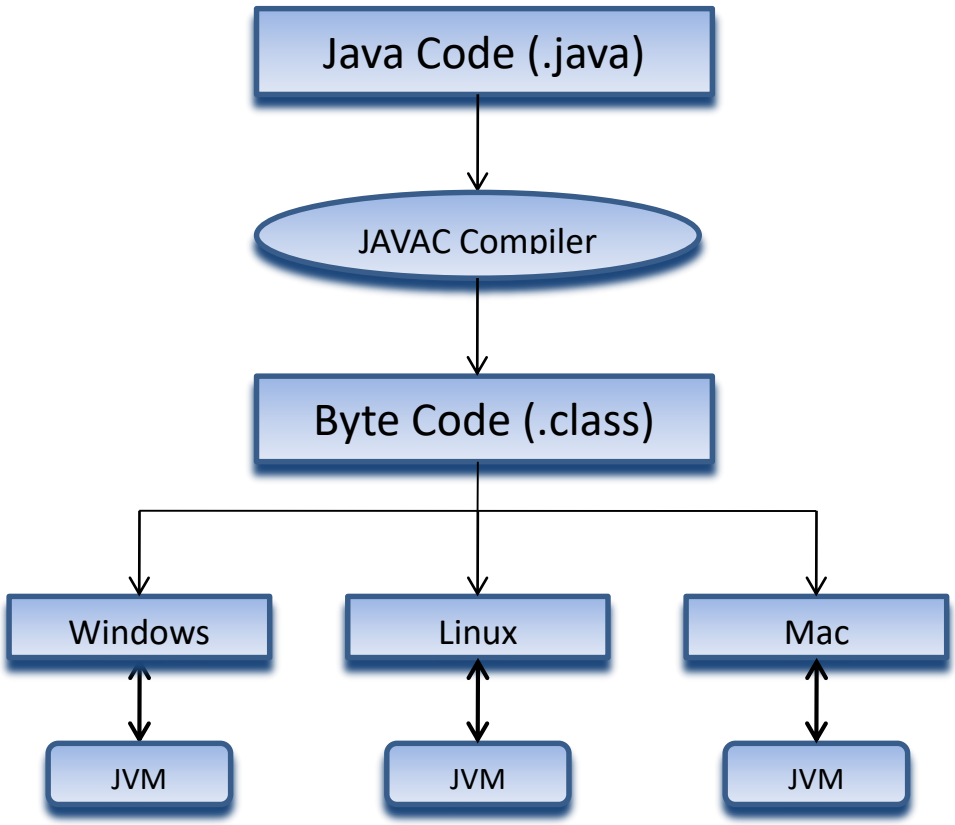
- Java is a programming language that:
- Is exclusively object oriented
- Has full GUI support
- Has full network support
- Is platform independent
- Executes stand-alone or “on-demand” in web browser as applets

JDK, JRE, Byte code & JVM.



- **Java Development Kit (JDK)**
  - JDK contains tools needed ,
    - To develop the Java programs and
    - JRE to run the programs.
  - The tools include compiler (javac.exe), Java application launcher (java.exe), Appletviewer, etc…
  - Java application launcher (java.exe),
    - Opens a JRE, loads the class, and invokes its main method.
- **Java Runtime Environment (JRE)**
  - The Java Runtime Environment (JRE) is required to run java applications.
  - It combines the Java Virtual Machine (JVM), platform core classes and supporting libraries.
  - JRE is part of the Java Development Kit (JDK), but can be downloaded separately.
  - It does not contain any development tools such as compiler, debugger, etc.
- **Byte code**
  - Byte code is intermediate representation of java source code.
  - It produce by java compiler by compiling java source code.
  - Extension for java class file or byte code is '.class'.
  - It is platform independent.

- **JVM (Java Virtual Machine)**
  - JVM is virtual because It provides a machine interface that does not depend on the operating system and machine hardware architecture.
  - JVM interprets the byte code into the machine code.
  - JVM itself is platform dependent, but Java is Not.



Explain features of JAVA.

Features of java are discussed below:

- The Features of Java programming are as below

1. Simple
2. Secure
3. Portable
4. Object-oriented
5. Robust
6. Multithreaded
7. Architecture-natural
8. Interpreted
9. High performance
10. Distributed
11. Dynamic

- **Simple**
  - It's simple because it contains many features of other languages like C and C++
  - It also removed complexities like pointers, Storage classes, goto statement and multiple Inheritance.
- **Secure**
  - Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:
    1. No explicit pointer
    2. Java Programs run inside virtual machine sandbox
    3. Bytecode Verifier
- **Portable**
  - Java is portable because it facilitates you to carry the java bytecode to any platform.
- **Object oriented**
  - Java is Object-oriented programming language. Everything in Java is an object.
- **Robust**
  - Robust simply means strong. Java is robust because:
    1. It uses strong memory management.
    2. There are lack of pointers that avoids security problem.
    3. There is automatic garbage collection in java.
    4. There is exception handling and type checking mechanism in java. All these points makes java robust
- **Multithreaded**
  - A thread is like a separate program, executing concurrently.
  - We can write Java programs that deal with many tasks at once by defining multiple threads.
  - The main advantage of multi-threading is that it doesn't occupy memory for each thread.
  - It shares a common memory area. Threads are important for multi-media, Web applications etc…
- **Architecture-neutral**
  - Java is architecture neutral because there is no implementation dependent features e.g. size of primitive types is fixed.
  - Example : in c int occupy 2 byte for 32 bit OS and 4 bytes for 64 bit OS whereas in JAVA it occupy 4 byte for int both in 32 bit and 64 bit OS.
- **Interpreted**
  - Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode.
  - This code can be executed on any system that implements the Java Virtual Machine.
- **High-Performance**
  - Most previous attempts at cross-platform solutions have done so at the expense of performance.
  - As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.
- **Dynamic**
  - Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.
  - This makes it possible to dynamically link code in a safe and expedient manner.

- **Distributed**
  - Java is distributed because it facilitates us to create distributed applications in java.
  - RMI and EJB are used for creating distributed applications.
  - We may access files by calling the methods from any machine on the internet.
- **Platform Independent**
  - Java is a platform independent programming language, because when you install JDK in the system then JVM is also installed automatically on the system.
  - For every operating system separate JVM is available which is capable to read the .class file or byte code.
  - When we compile Java code then .class file is generated by java compiler (javac) these codes are readable by the JVM and every operating system have its own JVM so JVM is platform dependent but due to JVM java is platform independent.

Explain Operators in JAVA

Sr.	Operator	Examples
1	Arithmetic Operators	+, -, *, /, %
2	Relational Operators	<, <=, >, >=, ==, !=
3	Logical Operators	&&, ||, !
4	Assignment Operators	=, +=, -=, *=, /=
5	Increment and Decrement Operators	++, --
6	Conditional Operator	?:
7	Bitwise Operators	&, |, ^, <<, >>

Arithmetic Operator

- An arithmetic operator performs basic mathematical calculations such as addition, subtraction, multiplication, division etc. on numerical values (constants and variables).
- **Increment / Decrement Operators**
  - Increment and decrement operators are unary operators that add or subtract one, to or from their operand.
  - the increment operator ++ increases the value of a variable by 1, e.g. a++ means a=a+1
  - the decrement operator -- decreases the value of a variable by 1. e.g. a— means a=a–1
  - If ++ operator is used as a prefix (++a) then the value of a is incremented by 1 first then it returns the value.
  - If ++ operator is used as a postfix (a++) then the value of a is returned first then it increments value of a by 1.

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

Arithmetic Operators in JAVA, consider A as 10 & B as 20

Expression	Evaluation (Let's say a=10, c=15)
b = a++	Value of b would be 10 and value of a would be 11.
b = ++a	Value of b & a would be 11.
b = a--	Value of b would be 10 and value of a would be 9.
b = --a	Value of b & a would be 9.

Increment / Decrement Operators

Relational Operators

- A relational operators are used to compare two values.
- They check the relationship between two operands, if the relation is true, it returns 1; if the relation is false, it returns value 0.
- Relational expressions are used in decision statements such as if, for, while, etc

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Relational Operators in JAVA, consider A as 10 & B as 20

Bitwise Operators

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

Bitwise Operators in JAVA, consider A as 60 & B as 13

Logical Operators

- Logical operators are decision making operators.
- They are used to combine two expressions and make decisions.
- An expression containing logical operator returns either 0 or 1 depending upon whether expression results false or true.

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Logical Operators in JAVA, consider A as true & B as false

Assignment Operators

- Assignment operators are used to assign a new value to the variable.
- The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value or a result of an expression.
- Meaning of = in Maths and Programming is different.
    - Value of LHS & RHS is always same in Math.
    - In programming, value of RHS is assigned to the LHS

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Assignment Operators in JAVA

Operator Precedence & Associativity

- How does java evaluate 1 + 10 * 9 ?
  (1 + 10) * 9 = 99 OR 1 + (10 * 9) = 91
- To get the correct answer for the given problem Java came up with Operator precedence. (multiplication have higher precedence than addition so correct answer will be 91 in this case)
- For Operator, associativity means that when the same operator appears in a row, then to which direction the expression will be evaluated.

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

- Operators Precedence and Associativity are two characteristics of operators that determine the evaluation order of sub-expressions in absence of brackets.
- Operator precedence determines which operation is performed first in an expression with more than one operators with different precedence.
- a=10 + 20 * 30 is calculated as 10 + (20 * 30) and not as (10 + 20) * 30 so answer is 610.
- Operators Associativity is used when two operators of same precedence appear in an expression. Associativity can be either Left to Right (L to R) or Right to Left (R to L).
- E.g. a=100 / 10 * 10
  If Left to Right means (100 / 10) * 10 then answer is 100
  If Right to Left means 100 / (10 * 10) then answer is 1
  Division (/) & Multiplication (*) are Left to Right associative so the answer is 100.

Explain short circuit operators.

- Java provides two interesting Boolean operators not found in many other computer languages. These are secondary versions of the Boolean AND and OR operators, and are known as shortcircuit logical operators.
- The OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is. If you use the || and && forms, rather than the | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.
- This is very useful when the right-hand operand depends on the value of the left one in order to function properly. For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

  if (denom != 0 && num / denom > 10)
- Since the short-circuit form of AND (&&) is used, there is no risk of causing a run-time exception when denom is zero. If this line of code were written using the single & version of AND, both sides would be evaluated, causing a run-time exception when denom is zero. It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:

  if(c==1 & e++ < 100) d = 100)
- Here, using a single & ensures that the increment operation will be applied to e whether c is equal to 1 or not.

Explain primitive Data types of JAVA.

Java defines 8 primitive types:

Data Type	Size	Range	Example
byte	1 Byte	-128 to 127	byte a = 10;
short	2 Bytes	-32,768 to 32,767	short a = 200;
int	4 Bytes	-2,147,483,648 to 2,147,483,647	int a = 50000;
long	8 Bytes	-9,223,372,036,854,775,80 to 9,223,372,036,854,775,807	long a = 20;
float	4 Bytes	1.4e-045 to 3.4e+038	float a = 10.2f;
double	8 Bytes	4.9e-324 to 1.8e+308	double a = 10.2;
char	2 Bytes	0 to 65536 (Stores ASCII of character)	char a = 'a';
boolean	Not defined	true or false	boolean a = true;

- **byte**
  - Smallest integer type
  - It is a signed 8-bit type (**1 Byte**)
  - Range is -128 to 127
  - Especially useful when working with stream of data from a network or file
  - Example: byte b = 10;
- **short**
  - short is signed 16-bit (**2 Byte**) type
  - Range : -32768 to 32767
  - It is probably least used Java type
  - Example: short vId = 1234;
- **int**
  - The most commonly used type
  - It is signed 32-bit (**4 Byte**) type
  - Range: -2,147,483,648 to 2,147,483,647
  - Example: int a = 1234;
- **long**
  - long is signed 64-bit (**8 Byte**) type
  - It is useful when int type is not large enough to hold the desired value
  - Example: long soconds = 1234124231;
- **char**
  - It is 16-bit (**2 Byte**) type
  - Range: 0 to 65,536
  - Example: char first = 'A'; char second = 65;
- **float**
  - It is 32-bit (**4-Byte**) type
  - It specifies a single-precision value
  - Example: float price = 1234.45213**f**
- **double**
  - It uses 64-bit (**8-Byte**)
  - All math functions such as sin(),cos(),sqrt() etc… returns double value
  - Example: double pi = 3.14141414141414;
- **boolean**
  - The boolean data type has only two possible values: true and false.
  - This data type represents one bit of information, but its "size" isn't something that's precisely defined.

Explain Variable in Java

- The variable is the basic unit of storage in a java program.
- A variable is defined by the combination of identifiers, a type and an optional initialize.
- All variables have a scope, which defines their visibility and a life time.
- Naming Rules for a variable: -
  - It should start with a lowercase letter such as id, name.
  - It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
  - If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
  - Avoid using one-character variables such as x, y, z.
- Declaring a Variable: -
  - All variables must be declared before they can be used. The basic form of a variable declaration is shown here.
  - Type identifier [= value] [, identifier [=value]...];
  - Int a,b,c; // declare 3 integers
  - Byte z = 22; // initialize z
  - Char x = 'X'; // the variable x has the value 'X'

Escape Sequences

- Escape sequences in general are used to signal an alternative interpretation of a series of characters.
- For example, if you want to put quotes within quotes you must use the escape sequence, \", on the interior quotes.
  System.out.println("Good Morning \"World\" ");

Escape Sequence	Description
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New Line
\t	Tab

Program Structure, Compilation and Run Process

- **Simple Java Program Structure**

```
public class Example
{
      public static void main(String args[])
      {
            System.out.println("First Example");
      }
}
```

- **class Example**

  Here name of the class is Example.

- **public static void main(String args[])**
  - public: The public keyword is an access specifier, which means that the content of the following block accessible from all other classes.
  - static: The keyword static allows main() to be called without having to instantiate a particular instance of a class.
  - void: The keyword void tells the compiler that main() does not return a value. The methods can return value.
  - main(): main is a method called when a java application begins,
  - String args []

    Declares a parameter named args, which is an array of instance of the class string.

    Args[] receives any command-line argument present when the program is executed.

- **System.out.println()**
  - System is predefined class that provides access to the system.
  - Out is the output stream that is connected to the console.
  - Output is accomplished by the built-in println() method. Println() displays the string which is passed to it.

- **Compilation of Java Program**

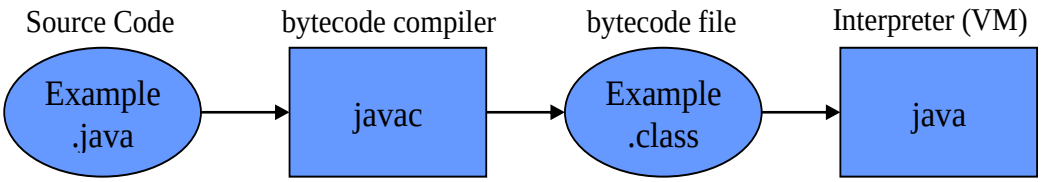


Figure 1.4. Java Program Compilation Process

- **Command 1:** Javac Example.java
  This command will compile the source file and if the compilation is successful, it will generate a file named example.class containing bytecode. Java compilers translate java program to bytecode form.
- **Command 2**: Java Example
  The command called 'java' takes the bytecode and runs the bytecode on JVM
- **Output**
  First Example

Type Casting

- Assigning a value of one type to a variable of another type is known as Type Casting.
- In Java, type casting is classified into two types,
- Widening/Automatic Type Casting (Implicit)

Widening/Automatic Type Casting (Implicit)

Narrowing Type Casting (Explicitly done)

Automatic Type Casting

- When one type of data is assigned to other type of variable , an automatic type conversion will take place if the following two conditions are satisfied:
- The two types are compatible
- The destination type is larger than the source type
- Such type of casting is called "widening conversion".
- Example:
  int can always hold values of byte and short

```
public static void main(String[] args) {
      byte b = 5;
      // √ this is correct
      int a = b;
}
```

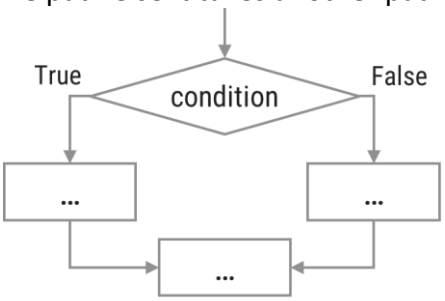
Casting Incompatible Types

- To create a conversion between two incompatible types, you must use a cast
- A cast is an explicit type conversion.
- Such type is called “narrowing conversion”.
- Syntax:(target-type) value
- Example:

```
public static void main(String[] args) {
      int a = 5;
      // × this is not correct
      byte b = a;
      // √ this is correct
      byte b = (byte)a ;
}
```

Decision Making Statements

- Compiler executes program statements sequentially.
- Decision making statements are used to control the flow of program execution.
- It allows us to control whether a set of program statement should be executed or not.
- It evaluates condition or logical expression first and based on its result (true or false), the control is transferred to the particular statement.
- If result is true then it takes one path else it takes another path.

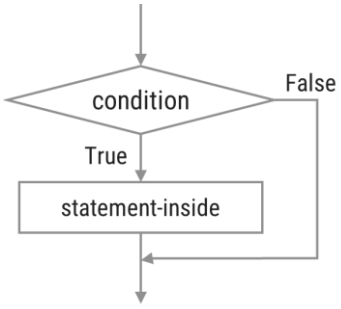


Commonly used decision making statements are:

1. One way Decision: if (Also known as simple if)
2. Two way Decision: if…else
3. Multi way Decision: if…else if…else if…else
4. Decision within Decision: nested if
5. Two way Decision: ?: (Conditional Operator)
6. n-way Decision: switch…case

If Statement

- if statement is the most simple decision-making statement also known as simple if.
- if statement consists of a Boolean expression followed by one or more statements.
- If the expression is true, then 'statement-inside' will be executed, otherwise 'statement-inside' is skipped and only 'statement-outside' will be executed.
- It is used to decide whether a block of statements will be executed or not i.e. if a certain condition is true then a block of statement is executed otherwise not.




```
if(condition)
{
        // Statements to execute if condition is true
}
```

WAP to print if a number is positive

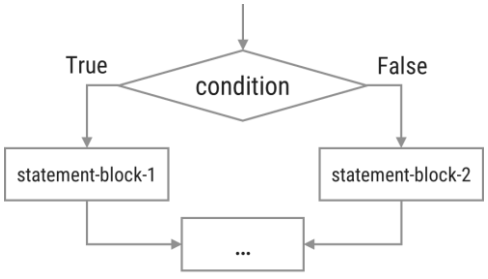
```
1. import java.util.*;
2. class MyProgram{
3. public static void main (String[] args){
4. int x;
5. Scanner sc = new Scanner(System.in);
6.      x = sc.nextInt();
7.      if(x > 0){
8.        System.out.println("number is a positive");
9.      }
10.       }
```

WAP to print if a number is odd or even

```
1. import java.util.*;
2. class MyProgram{
3. public static void main (String[] args){
4. int x;
5. Scanner sc = new Scanner(System.in);
6.      x = sc.nextInt();
7.      if( x % 2 == 1 ){
8.        System.out.println("number is a odd");
9.      }
10.       if( x % 2 == 0 ){
11.            System.out.println("number is a even");
12.          }
13.       }
14.
```

If…else: Two way Decision

- For a simple if, if a condition is true, the compiler executes a block of statements, if condition is false then it doesn't do anything.
- What if we want to do something when the condition is false? if…else is used for the same.
- If the 'expression' is true then the 'statement-block-1' will get executed else 'statement-block-2' will be executed




```
if(condition)
{
      // statement-block-1
      // to execute if condition is
      true
}
else
{
      // statement-block-2
      // to execute if condition is
      false
}
```

WAP to print if a number is positive

```
1. import java.util.*;
2. class MyProgram{
3. public static void main (String[] args){
4. int x;
5. Scanner sc = new Scanner(System.in);
6. x = sc.nextInt();
7. if (x > 0){
          System.out.println("Number is positive");
8. }//if
          else{
9. System.out.println("Number is negative");
          }//else
          }//main
10.       }//class
```

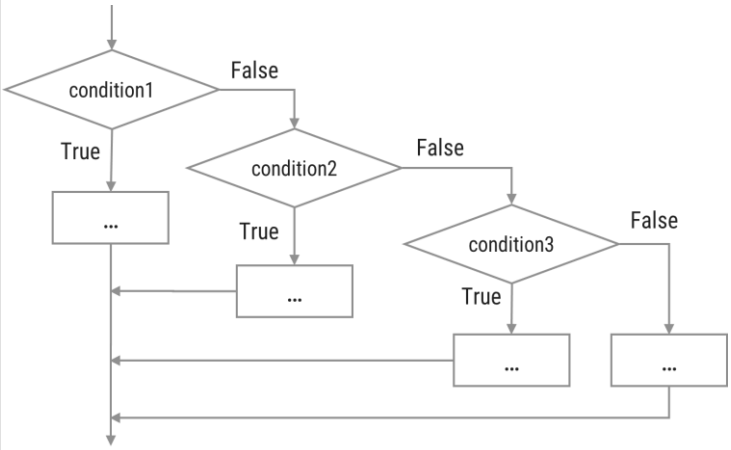
WAP to print if a number is odd or even

```
1. import java.util.*;
2. class MyProgram{
3. public static void main (String[] args){
4.        int x;
5.        Scanner sc = new Scanner(System.in);
6.      x = sc.nextInt();
7.      if( x % 2 == 1 ){
8.        System.out.println("number is a odd");
9.      }
10.       else{
          System.out.println("number is a even");
11.           }
12.       }
13.
```

if-else-if ladder

- if…else if…else statement is also known as if-else-if ladder which is used for multi way decision making.
- It is used when there are more than two different conditions.
- It tests conditions in a sequence, from top to bottom.
- If first condition is true then the associated block with if statement is executed and rest of the conditions are skipped.
- If condition is false then then the next if condition will be tested, if it is true then the associated block is executed and rest of the conditions are skipped. Thus it checks till last condition.
- Condition is tested only and only when all previous conditions are false.
- The last else is the default block which will be executed if none of the conditions are true.
- The last else is not mandatory. If there are no default statements then it can be skipped.

```
if(condition 1)
{
      statement-block1;
}
else if(condition 2)
{
       statement-block2;
}
else if(condition 3)
{
       statement-block3;
}
else if(condition 4)
{
      statement-block4;
}
else
      default-statement;
```



WAP to print if a number is zero or positive or negative

```
1. import java.util.*;
2. class MyProgram{
3. public static void main (String[] args){
4.       int x;
5.       Scanner sc = new Scanner(System.in);
6.      x = sc.nextInt();
7.       if(x > 0){
8.             System.out.println(" number is a positive");
9.       }
10.       else if(x < 0) {
11.                  System.out.println(" number is a negative");
12.            }
13.       else{
14.                  System.out.println(" number is a zero");
15.       }
16.      }
```

WAP to print day name from day number

```
1. public class Demo  {
2.   public static void main(String[] args)   {
3.       int d;
4.     Scanner sc = new Scanner(System.in);
5.       d = sc.nextInt();
6.       if (d == 1 )            System.out.println("Monday");
7.       else if (d == 2)  System.out.println("Tuesday");
8.       else if (d == 3)  System.out.println("Wednesday");
9.       else if (d == 4)  System.out.println("Thursday");
10.      else if (d == 5)  System.out.println("Friday");
11.      else if (d == 6)  System.out.println("Saturday");
12.      else                    System.out.println("Sunday");
13.      }
14. }
```

Nested If statement

- A nested if is an if statement that is the target of another if statement.
- Nested if statements mean an if statement inside another if statement.
- The statement connected to the nested if statement is only executed when -:
    - Condition of outer if statement is true, and
    - Condition of the nested if statement is also true.
- Note: There could be an optional else statement associated with the outer if statement, which is only executed when the condition of the outer if statement is evaluated to be false and in this case, the condition of nested if condition won't be checked at all

```
if(condition 1)
{
       if(condition 2)
       {
              nested-block;
       }
       else
       {
              nested-block;
       }
}//if
else if(condition 3)
{
        statement-block3;
}
else(condition 4)
{
       statement-block4;
}
```

Nested If Program

```
1. int username = Integer.parseInt(args[0]);
2. int password = Integer.parseInt(args[1]);
3. double balance = 123456.25;
4. if(username==1234){
5.       if(password==987654){
6.             System.out.println("Your Balance is ="+balance);
7.       }//inner if
8.       else{
9.             System.out.println("Password is invalid");
10.      }
   }//outer if
11.  else{
12.       System.out.println("Username is invalid");
13.  }
14.
```

Switch case

n-way Decision

- switch…case is a multi-way decision making statement.
- It is similar to if-else-if ladder statement.
- It executes one statement from multiple conditions.

```
switch (expression)
{
   case constant 1:
       // Statement-1
       break;
   case constant 2:
       // Statement-2
       break;
   case constant 3:
       // Statement-3
       break;
default:
      // Statement-default
//  if none of the above case matches then this block would be executed.
}
```

WAP to print day based on number entered

```
1. public class Demo  {
2.  public static void main(String[] args){
3.  int d;
4.  Scanner sc= new  Scanner(System.in);
5.  d = sc.nextInt();
6.  switch (d) {
7.      case 1:
8.          System.out.println("Monday"); break;
9.      case 2:
10.           System.out.println("Tuesday"); break;
11.       case 3:
12.           System.out.println("Wednesday"); break;
13.       case 4:
14.           System.out.println("Thursday"); break;
15.       case 5:
16.           System.out.println("Friday"); break;
17.       case 6:
18.           System.out.println("Saturday"); break;
19.       case 7:
20.           System.out.println("Sunday"); break;
21.       default:
22.          System.out.println("Invalid Day");
23.       } //switch
24.    }
25.  }
```

WAP to print day based on number entered

```
1. public class SwitchExampleDemo {
2. public static void main(String[] args)
3. {
4. int number = 20;
5. switch (number) {
6.    case 10:
7.       System.out.println("10");
8.       break;
9.    case 20:
10.      System.out.println("20");
11.      break;
12.   default:
13.      System.out.println("Not 10 or 20");
14.   }//switch
15.   }
16.   }
```

Points to remember for switch case

- The condition in the switch should result in a constant value otherwise it would be invalid.
- In some languages, switch statements can be used for integer values only.
- Duplicate case values are not allowed.
- The value for a case must be of the same data type as the variable in the switch.
- The value for a case must be a constant.
- Variables are not allowed as an argument in switch statement.
- The break statement is used inside the switch to terminate a statement sequence.
- The break statement is optional, if eliminated, execution will continue on into the next case.
- The default statement is optional and can appear anywhere inside the switch block.

Introduction to loop

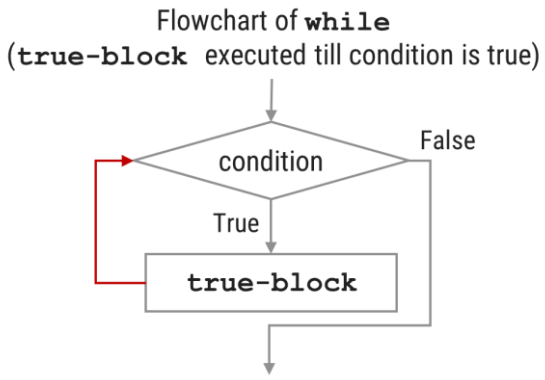
Repeatedly execute a block of statements

Looping Statements

- Sometimes we need to repeat certain actions several times or till the some criteria is satisfied.
- Loop constructs are used to iterate a block of statements several times.
- Loop constructs repeatedly execute a block of statements for a fixed number of times or till some condition is satisfied
- Following are looping statements in any programming language,
    - Entry Controlled **while, for**
    - Exit Controlled **do…while**
    - Unconditional Jump **goto** (It is advised to never use **goto** in a program)

Entry Controlled Loop: While

- **While** is an entry controlled loop.
- It executes a block of statements till the condition is true.




```
while(condition)
{
      // true-block
}
```

```
int i = 1;
while (i <= 5)
{ System.out.println(i);
  i++;
}
```

- If the number of iteration is not fixed, it is recommended to use while loop.

```
//code will print 1 to 9
   1. public class WhileLoopDemo {
   2. public static void main(String[] args) {
   3. int number = 1;
   4.       while(number < 10) {
   5.             System.out.println(number);
   6.             number++;
   7.       }
   8.    }
   9. }
```

WAP to print day based on number entered

```
1. public class SwitchExampleDemo {
2. public static void main(String[] args)
3. {
4. int number = 20;
5. switch (number) {
6.    case 10:
7.       System.out.println("10");
8.       break;
9.    case 20:
10.      System.out.println("20");
11.      break;
12.   default:
13.      System.out.println("Not 10 or 20");
14.   }//switch
15.   }
16.   }
```

WAP to print odd numbers between 1 to n

```
1. import java.util.*;
2. class WhileDemo{
3. public static void main (String[] args){
4.  int n,i=1;
5.  Scanner sc = new Scanner(System.in);
6.  System.out.print("Enter a number:");
7.  n = sc.nextInt();
8.  while(i <= n){
9.    if(i%2==1)
10.       System.out.println(i);
11.     i++;
12.   }
13.  }}
```

WAP to print factors of a given number

```
1. import java.util.*;
2. class WhileDemo{
3. public static void main (String[] args){
4. int i=1,n;
5. Scanner sc = new Scanner(System.in);
6. System.out.print("Enter a Number:");
7. n = sc.nextInt();
8. System.out.print(" Factors:");
9. while(i <= n){
10.     if(n%i == 0)
11.       System.out.print(i +",");
12.     i++;
13.   }
14.  }}
```

Entry Controlled Loop: for (;;) Loop

- for is an entry controlled loop
- Statements inside the body of for are repeatedly executed till the condition is true

```
for (initialization; condition; increment/decrement)
{
    // statements
}
```

```
for(i=1; i <= 5; i++)
{
   System.out.print("Hello World!");
}
```

- The initialization statement is executed only once, at the beginning of the loop.
- Then, the condition is evaluated.
  - If the condition is true, statements inside the body of for loop are executed
  - If the condition is false, the for loop is terminated.
- Then, increment / decrement statement is executed
- Again the condition is evaluated and so on so forth till the condition is true.
- If the number of iteration is fixed, it is recommended to use for loop.

```
//code will print 1 to 9
   1. public class ForLoopDemo {
   2. public static void main(String[] args){
   3.       for(int number=1;number<10;number++)
   4.       {
   5.             System.out.println(number);
   6.       }
   7.    }
   8. }
```

WAP to print odd numbers between 1 to n

```
1. import java.util.*;
2. class MyProgram{
3. public static void main (String[] args){
4.    int i=1;
5.    Scanner sc = new Scanner(System.in);
6.     n = sc.nextInt();
7.    for(i=1; i<=n; i++)  {
8.              if(i%2==1)
9.              System.out.println(i);
10.       }//for
11.     }//
12.   }
```

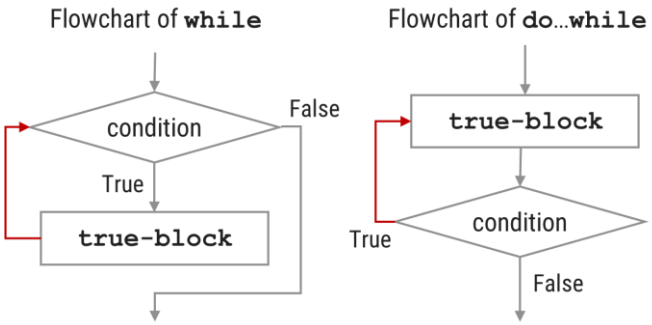
WAP to print factors of a given number

```
1. import java.util.*;
2. class MyProgram{
3. public static void main (String[] args){
4.    int i=1;
5.    Scanner sc = new  Scanner(System.in);
6.     n = sc.nextInt();
7.    for(i=1; i<=n; i++){
8.        if(n%i == 0)
9.          System.out.println(i);
10.     }
11.    }
12.   }
```

Exit Controlled Loop: do…while

- **do…while** is an exit controlled loop.
- do-while loop is executed at least once because condition is checked after loop body.
- Statements inside the body of **do…while** are repeatedly executed till the condition is true.
- **while** loop executes zero or more times, **do…while** loop executes one or more times.

```
do
{
        // true-block
}
while(condition) ;
```




```
//code will print 1 to 9
   1. public class DoWhileLoopDemo {
   2. public static void main(String[] args) {
   3.       int number = 1;
   4.       do {
   5.             System.out.println(number);
   6.             number++;
   7.       }while(number < 10) ;
   8.   }
   9. }
```

WAP to print 1 to 10 using do-while loop

```
1. import java.util.*;
2. class MyProgram{
3. public static void main (String[] args){
4.    int i=1;
5.    Scanner sc = new  Scanner(System.in);
6.     n = sc.nextInt();
7.    for(i=1; i<=n; i++){
8.        if(n%i == 0)
9.          System.out.println(i);
10.     }
11.    }
12.   }
```

Continue: Skip the statement in the iteration

- Sometimes, it is required to skip the remaining statements in the loop and continue with the next iteration.
- continue statement is used to skip remaining statements in the loop.
- continue is keyword in java.

WAP to calculate the sum of positive numbers.

```
1. import java.util.*;
2. class ContinueDemo{
3. public static void main(String[] args) {
4. int a,n,sum=0;
5. Scanner sc = new Scanner(System.in);
6. n = sc.nextInt();
7. for(int i=0;i<n;i++){
8.   a = sc.nextInt();
9.   if(a<0){
10.      continue;
11.      System.out.println("a="+a); //error:unreachable
                                        statement
12.    }//if
13.    sum=sum+a;
14.  }//for
15.   System.out.println("sum="+sum);
16.   }
17.  }
```

Break: Early exit from the loop

- Sometimes, it is required to early exit the loop as soon as some situation occurs.
- E.g. searching a particular number in a set of 100 numbers. As soon as the number is found it is desirable to terminate the loop.
- *break* is keyword in java.
- *break* statement is used to jump out of a loop.
- *break* statement provides an early exit from for, while, do…while and switch constructs.
- *break* causes exit from the innermost loop or switch.

WAP to calculate the sum of given numbers. User will enter -1 to terminate.

```
1. import java.util.*;
2. class BreakDemo{
3. public static void main (String[] args){
4.     int a,sum=0;
5.     System.out.println("enter numbers_ enter -1 to break");
6.     Scanner sc = new Scanner(System.in);
7.     while(true){
8.         a = sc.nextInt();
9.         if(a==-1)
10.              break;
11.          sum=sum+a;
12.       }//while
13.       System.out.println("sum="+sum);
14.    }
15.  }
```

Nested loop

loop within a loop

Pattern Programs

WAP to print given pattern (nested loop)

```
1. public static void main(String[] args) {
2. int n=5;
3. for(int i=1;i<=n;i++){
4.       for(int j=1;j<=i;j++){
5.             System.out.print("*");
6.       }//for j
7.       System.out.println();
8.      }//outer for i
9.     }
10.  }
```

```
*
**
***
****
*****
```

WAP to print given pattern (nested loop)

```
1. class PatternDemo{
2. public static void main(String[] args) {
3.    int n=5;
4.    for(int i=1;i<=n;i++){
5.       for(int j=1;j<=i;j++){
                System.out.print(j+"\t");
6.       }//for j
7.        System.out.println();
8.     }//outer for i
9.     }
10.  }
```

Mathematical functions

- The Java Math class provides more advanced mathematical calculations other than arithmetic operator.
- The java.lang.Math class contains methods which performs basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
- All the methods of class Math are static.
- Fields :
    - Math class comes with two important static fields
        - E : returns double value of Euler's number (i.e 2.718281828459045).
        - PI : returns double value of PI (i.e. 3.141592653589793).

Math class Method:

Method	Description
Math.abs()	It will return the Absolute value of the given value.
Math.max()	It returns the Largest of two values.
Math.min()	It is used to return the Smallest of two values.
Math.round()	It is used to round of the decimal numbers to the nearest value.
Math.sqrt()	It is used to return the square root of a number.
Math.cbrt()	It is used to return the cube root of a number.
Math.pow()	It returns the value of first argument raised to the power to second argument.
Math.signum()	It is used to find the sign of a given value.
Math.ceil()	It is used to find the smallest integer value that is greater than or equal to the argument or mathematical integer.
Math.copySign()	It is used to find the Absolute value of first argument along with sign specified in second argument.
Math.nextAfter()	It is used to return the floating-point number adjacent to the first argument in the direction of the second argument.
Math.nextUp()	It returns the floating-point value adjacent to d in the direction of positive infinity.
Math.nextDown()	It returns the floating-point value adjacent to d in the direction of negative infinity.
Math.floor()	It is used to find the largest integer value which is less than or equal to the argument and is equal to the mathematical integer of a double value.
Math.floorDiv()	It is used to find the largest integer value that is less than or equal to the algebraic quotient.
Math.random()	It returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
Math.rint()	It returns the double value that is closest to the given argument and equal to mathematical integer.
Math.hypot()	It returns sqrt(x^2 +y^2) without intermediate overflow or underflow.
Math.ulp()	It returns the size of an ulp of the argument.
Math.getExponent()	It is used to return the unbiased exponent used in the representation of a value.
Math.IEEEremainder()	It is used to calculate the remainder operation on two arguments as prescribed by the IEEE 754 standard and returns value.
Math.addExact()	It is used to return the sum of its arguments, throwing an exception if the result overflows an int or long.

Math.subtractExact()	It returns the difference of the arguments, throwing an exception if the result overflows an int.
Math.multiplyExact()	It is used to return the product of the arguments, throwing an exception if the result overflows an int or long.
Math.incrementExact()	It returns the argument incremented by one, throwing an exception if the result overflows an int.
Math.decrementExact()	It is used to return the argument decremented by one, throwing an exception if the result overflows an int or long.
Math.negateExact()	It is used to return the negation of the argument, throwing an exception if the result overflows an int or long.
Math.toIntExact()	It returns the value of the long argument, throwing an exception if the value overflows an int.

Logarithmic Math Method

Method	Description
Math.log()	It returns the natural logarithm of a double value.
Math.log10()	It is used to return the base 10 logarithm of a double value.
Math.log1p()	It returns the natural logarithm of the sum of the argument and 1.
Math.exp()	It returns E raised to the power of a double value, where E is Euler's number and it is approximately equal to 2.71828.
Math.expm1()	It is used to calculate the power of E and subtract one from it.

Trigonometric Math Method

Method	Description
Math.sin()	It is used to return the trigonometric Sine value of a Given double value.
Math.cos()	It is used to return the trigonometric Cosine value of a Given double value.
Math.tan()	It is used to return the trigonometric Tangent value of a Given double value.
Math.asin()	It is used to return the trigonometric Arc Sine value of a Given double value
Math.acos()	It is used to return the trigonometric Arc Cosine value of a Given double value.
Math.atan()	It is used to return the trigonometric Arc Tangent value of a Given double value.

Math Example

```
1. public class MathDemo {
2. public static void main(String[] args) {
3.    double sinValue = Math.sin(Math.PI / 2);
4.    double cosValue = Math.cos(Math.toRadians(80));
5.    int randomNumber = (int)(Math.random() * 100);
6.    // values in Math class must be given in Radians
7.    // (not in degree)
8.        System.out.println("sin(90) = " + sinValue);
9.        System.out.println("cos(80) = " + cosValue);
10.       System.out.println("Random = " + randomNumber);
11.     }
12.   }
```

Arrays in Java

Definition: *An array is a fixed size sequential collection of elements of same data type grouped under single variable name.*

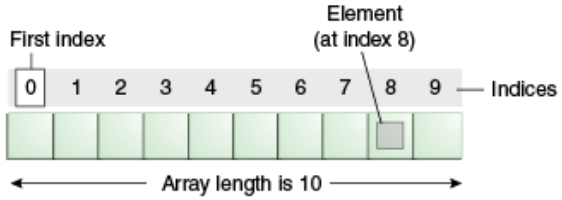
- An array is a group of like-typed variables that are referred by a common name.
- Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index.
- Arrays offer a convenient means of grouping related information.

```
int percentage[10];
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

- Often we need to deal with relatively large set of data.
- E.g.
  - Percentage of all the students of the college. (May be in thousands)
  - Age of all the citizens of the city. (May be lakhs)
- We need to declare thousands or lakhs of the variable to store the data which is practically not possible.
- We need a solution to store more data in a single variable.
- Array is the most appropriate way to handle such data.
- As per English Dictionary, "Array means collection or group or arrangement in a specific order."

Array declaration



Array declaration:

type var-name[];

Example:

Int student_marks[];

Above example will represent array with no value (null) To link student_marks with actual, physical array of integers, we must allocate one using new keyword.

Example:

int student_marks[] = new int[20];

- Normal Variable Declaration: int a;
- Array Variable Declaration: int b[10];
- Individual value or data stored in an array is known as an element of an array.
- Positioning / indexing of an elements in an array always starts with 0 not 1.
  - If 10 elements in an array then index is 0 to 9
  - If 100 elements in an array then index is 0 to 99
  - If 35 elements in an array then index is 0 to 34

- Variable a stores 1 integer number where as variable b stores 10 integer numbers which can be accessed as b[0], b[1], b[2], b[3], b[4], b[5], b[6], b[7], b[8] and b[9].

Important point about Java array

- An array is **derived** datatype.
- An array is **dynamically** allocated.
- The individual elements of an array is refereed by their **index/subscript** value.
- The **subscript** for an array always begins with **0**.

One-Dimensional Array

- An array using one subscript to represent the list of elements is called one dimensional array.
- A One-dimensional array is essentially a list of like-typed variables.
- Array declaration: type var-name[];
- Example: int student_marks[];
- Above example will represent array with no value (null).
- To link student_marks with actual array of integers, we must allocate one using new keyword.
- Example:

int student_marks[] = new int[20];

35	13	28	106	35	42	5	83	97	14
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Example (One-Dimensional Array)

```
1. public class ArrayDemo{
2. public static void main(String[] args) {
3.       int a[]; // or int[] a
4. // till now it is null as it does not assigned any
                                                    memory
5.       a = new int[5]; // here we create an array
6.       a[0] = 5;
7.       a[1] = 8;
8.       a[2] = 15;
9.       a[3] = 84;
10.      a[4] = 53;
11.      /* in java we use length property to determine the
                                                  length
12.       * of an array, unlike c where we used sizeof
                                              function */
13.      for (int i = 0; i < a.length; i++) {
14.            System.out.println("a["+i+"]="+a[i]);
15.      }
16.   }
17.      }
```

Example (One-Dimensional Array)

```
1. Class AutoArray{
2.       public static void main (String args[ ])
3.       {
4.             int month_days[]={31,28,31,30,31,30,31,31,
5.             30,31,30,31};
6.             System.out.println("April has" +
                           month_days[3]     + "days.");
7.       }  }
```

WAP to store 5 numbers in an array and print them

```
1. import java.util.*;
2. class ArrayDemo1{
3. public static void main (String[] args){
4.       int i, n;
5.       int[] a=new int[5];
6.       Scanner sc = new Scanner(System.in);
7.       System.out.print("enter Array Length:");
8.       n = sc.nextInt();
9.       for(i=0; i<n; i++) {
10.            System.out.print("enter a["+i+"]:");
11.            a[i] = sc.nextInt();
12.      }
13.      for(i=0; i<n; i++)
14.          System.out.println(a[i]);
15.         }
16.      }
```

Output:

```
enter Array Length:5
enter a[0]:1
enter a[1]:2
enter a[2]:4
enter a[3]:5
enter a[4]:6
1
2
4
5
6
```

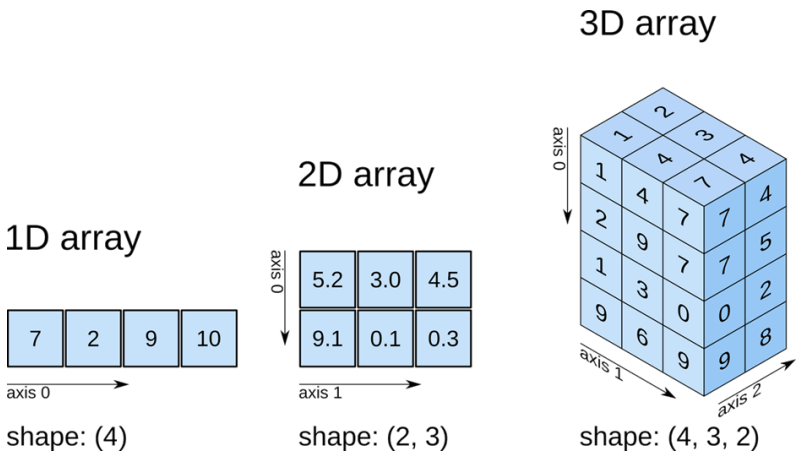
WAP to print elements of an array in reverse order

```
1. import java.util.*;
2. public class RevArray{
3. public static void  main(String[] args) {
4.   int i, n;
5.   int[] a;
6.   Scanner sc=new Scanner(System.in);
7.   System.out.print("Enter Size of an Array:");
8.   n=sc.nextInt();
9.   a=new int[n];
10.        for(i=0; i<n; i++){
11.            System.out.print("enter a["+i+"]:");
12.            a[i]=sc.nextInt();
13.        }
14.        System.out.println("Reverse Array");
15.        for(i=n-1; i>=0; i--)
16.            System.out.println(a[i]);
17.       }
18.      }
```

WAP to count positive number, negative number and zero from an array of n size

```
1. import java.util.*;
2. class ArrayDemo1{
3. public static void main (String[] args){
4. int n,pos=0,neg=0,z=0;
5. int[] a=new int[5];
6. Scanner sc = new Scanner(System.in);
7. System.out.print("enter Array Length:");
8. n = sc.nextInt();
9.       for(int i=0; i<n; i++) {
10.            System.out.print("enter a["+i+"]:");
11.            a[i] = sc.nextInt();
12.            if(a[i]>0)
13.                  pos++;
14.            else if(a[i]<0)
15.                  neg++;
16.            else
17.                  z++;
18.      }
19.      System.out.println("Positive no="+pos);
20.      System.out.println("Negative no="+neg);
21.      System.out.println("Zero no="+z);
22.      }}
```

Multidimensional Array



- In java, Multidimensional arrays are actually array of arrays.
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- For example, following declares two dimensional array:
  int twoD[ ] [ ] = new int [4] [5];
- This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int.
- Alternative Array Declaration, There is a second form that may be used to declare an array:
  type[ ] var-name;
- The square brackets follow the type specifier, and not the name of the array variable.
- For example, the following two declarations are equivalent:
  int a1[ ] = new int[4];
  int [ ] a1= new int[4];
  char twod [ ] [ ] = new char [3] [4];
  char [ ] [ ] twod = new char [3] [4];
- This alternative declaration form offers convenience when declaring several arrays at the same time. For example,
  int [ ] nums1, nums2, nums3;
- This creates 3 array variables of int type.
- Example:

```
int runPerOver[][] = new int[50][6];
Manually allocate different size:
Int runPerOver[][] = new int[3][];
runPerOver[0] = new int[6];
runPerOver[1] = new int[7];
runPerOver[2] = new int[6];
```

Initialization :

```
Int runPerOver[][] = {
                        {0,4,2,1,0,6},
                        {1,56,4,1,2,4,0},
                        {6,4,1,0,2,2},
```

WAP to read 3 x 3 elements in 2d array

```
1. import java.util.*;
2. class Array2Demo{
3. public static void main(String[] args) {
4.       int size;
5.       Scanner sc=new Scanner(System.in);
6.       System.out.print("Enter size of an array");
7.       size=sc.nextInt();
8.       int a[][]=new int[size][size];
9.       for(int i=0;i<a.length;i++){
10.            for(int j=0;j<a.length;j++){
11.                 a[i][j]=sc.nextInt();
12.            }
13.      }
14.      for(int i=0;i<a.length;i++){
15.            for(int j=0;j<a.length;j++){
16.                 System.out.print
                      ("a["+i+"]["+j+"]:"+a[i][j]+"\t");
17.            }
18.            System.out.println();
19.      }
20.    }
21.  }
```

Output:

```
11
12
13
14
15
16
17
18
19
a[0][0]:11      a[0][1]:12      a[0][2]:13
a[1][0]:14      a[1][1]:15      a[1][2]:16
a[2][0]:17      a[2][1]:18      a[2][2]:19
```

WAP to perform addition of two 3 x 3 matrices

```
1. import java.util.*;
2. class Array2Demo{
3. public static void main(String[] args) {
4.       int size;
5.       int a[][],b[][],c[][];
6.       Scanner sc=new Scanner(System.in);
7.       System.out.print("Enter size of an array:");
8.       size=sc.nextInt();
9.       a=new int[size][size];
10.       System.out.println("Enter array elements:");
11.      for(int i=0;i<a.length;i++){
12.            for(int j=0;j<a.length;j++){
13.                  System.out.print("Enter
                                        a["+i+"]["+j+"]:");
14.                  a[i][j]=sc.nextInt();
15.            }
16.      }
17.      b=new int[size][size];
18.      for(int i=0;i<b.length;i++){
19.       for(int j=0;j<b.length;j++){
20.         System.out.print("Enter b["+i+"]["+j+"]:");
21.         b[i][j]=sc.nextInt();
22.       }
23.      }
24.      c=new int[size][size];
25.      for(int i=0;i<c.length;i++){
26.       for(int j=0;j<c.length;j++){
27.            System.out.print("c["+i+"]["+j+"]:"
                                  +(a[i][j]+b[i][j])+"\t");
28.        }
29.        System.out.println();
30.        }//outer for
31.       }//main()
32.      }//class
```

Output:

```
Enter size of an array:3
Enter array elements:
Enter a[0][0]:1
Enter a[0][1]:1
Enter a[0][2]:1
Enter a[1][0]:1
Enter a[1][1]:1
Enter a[1][2]:1
Enter a[2][0]:1
Enter a[2][1]:1
Enter a[2][2]:1
Enter b[0][0]:4
Enter b[0][1]:4
Enter b[0][2]:4
Enter b[1][0]:4
Enter b[1][1]:4
Enter b[1][2]:4
```

```
Enter b[2][0]:4
Enter b[2][1]:4
Enter b[2][2]:4
c[0][0]:5       c[0][1]:5       c[0][2]:5
c[1][0]:5       c[1][1]:5       c[1][2]:5
c[2][0]:5       c[2][1]:5       c[2][2]:5
```

Initialization of an array elements

One dimensional Array

- int a[5] = { 7, 3, -5, 0, 11 }; // a[0]=7, a[1] = 3, a[2] = -5, a[3] = 0, a[4] = 11
- int a[5] = { 7, 3 }; // a[0] = 7, a[1] = 3, a[2], a[3] and a[4] are 0
- int a[5] = { 0 }; // all elements of an array are initialized to 0

Two dimensional Array

- int a[2][4] = { { 7, 3, -5, 10 }, { 11, 13, -15, 2} }; // 1st row is 7, 3, -5, 10 & 2nd row is 11, 13, -15, 2
- int a[2][4] = { 7, 3, -5, 10, 11, 13, -15, 2 }; // 1st row is 7, 3, -5, 10 & 2nd row is 11, 13, -15, 2
- int a[2][4] = { { 7, 3 }, { 11} }; // 1st row is 7, 3, 0, 0 & 2nd row is 11, 0, 0, 0
- int a[2][4] = { 7, 3 }; // 1st row is 7, 3, 0, 0 & 2nd row is 0, 0, 0, 0
- int a[2][4] = { 0 }; // 1st row is 0, 0, 0, 0 & 2nd row is 0, 0, 0, 0

Searching in Array

- Searching is the process of looking for a specific element in an array. for example, discovering whether a certain element is included in the array.
- Searching is a common task in computer programming. Many algorithms and data structures are devoted to searching.
- We will discuss two commonly used approaches as follows:

Linear Search

The linear search approach compares the key element key sequentially with each element in the array. It continues to do so until the key matches an element in the
array or the array is exhausted without a match being found.

LinearSearchDemo.java

```
1. import java.util.*;
2. class LinearSearchDemo{
3. public static void main(String[] args) {
4.       int size;
5.       int a[]={1,2,3,4,5,6,7,8,9};
6.       int search;
7.       boolean flag=false;
8.       Scanner sc=new Scanner(System.in);
9.       System.out.print("Enter element to search");
10.      search=sc.nextInt();
11.      for(int i=0;i<a.length;i++){
12.            if(a[i]==search){
13.                  System.out.println("element found at
                                          "+i+"th index");
14.                  flag=true;
15.                  break;
16.            } //if
17.      }
18.      if(!flag)
19.      System.out.println("element NOT found!");
20.      }
21.   }
```

Output:

```
Enter element to search 6
element found at 5th index
Enter element to search 35
element NOT found!
```

Binary Search

The binary search first compares the key with the element in the middle of the array. Consider the following three cases:

- If the key is less than the middle element, you need to continue to search for the key only in the first half of the array.
- If the key is equal to the middle element, the search ends with a match.
- If the key is greater than the middle element, you need to continue to search for the key only in the second half of the array.

Binary Search Demo

```
1. import java.util.*;
2. class BinaryDemo{
3. public static void main(String[] args){
4.   int size;
5.   int a[]={1,2,3,4,5,6,7,8,9};
6.   int search;
7.   boolean flag=false;
8.   Scanner sc=new Scanner(System.in);
9.   System.out.print("Enter element to search:");
10.  search=sc.nextInt();
11.  int low=0;
12.  int high= a.length-1;
13.  while(high>=low){
14.      int mid=(high+low)/2;
15.      if(search==a[mid]){
16.            flag=true;
17.            System.out.println("element found
                     at "+mid+" index ");
18.        break;
19.      }
20.      else if(search<a[mid]){
21.            high=mid-1;
22.      }
23.      else if(search>a[mid]){
24.            low=mid+1;
25.      }
26.    }//while

27.      if(!flag)
28.       System.out.println("element not found");
29.        }
30.      }
```

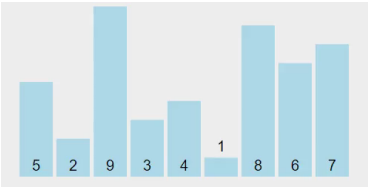
Output:

```
Enter element to search:5
element found at 4 index
Enter element to search:9
element found at 8 index
Enter element to search:56
element not found
```

***Note**: Array should be sorted in ascending order if we want to use Binary Search.*

Sorting Array

- Sorting, like searching, is a common task in computer programming. Many different algorithms have been developed for sorting.
- There are many sorting techniques available, we are going to explore selection sort.
- Selection sort
  - Finds the smallest number in the list and swaps it with the first element.
  - It then finds the smallest number remaining and swaps it with the second element, and so on, until only a single number remains.



Selection Sort Demo

```
1. import java.util.*;
2. class SelectionSortDemo{
3.  public static void main(String[] args) {
4.  int a[]={ 5, 2, 9, 3, 4, 1, 8, 6, 7 };
5. for (int i = 0; i < a.length - 1; i++) {
6. // Find the minimum in the list[i..a.length-1]
7.   int min = a[i];
8.   int minIndex = i;
9.   for (int j = i + 1; j < a.length; j++) {
10.         if (min > a[j]) {
11.           min = a[j];
12.           minIndex = j;
13.         }
14.        }//inner for loop j
15.      // Swap a[i] with a[minIndex]
16.        if (minIndex != i) {
17.            a[minIndex] = a[i];
18.            a[i] = min;
19.            }
20.       }//outer for i
21.       for(int temp: a) { // this is foreach loop
22.            System.out.print(temp + ", ");
23.         }
24.        }//main()
25.  }//class
Output:
1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Methods in Java

What is Method?

- A method is a group of statements that performs a specific task.
- A large program can be divided into the basic building blocks known as method/function.
- The function contains the set of programming statements enclosed by { }.
- Program execution in many programming language starts from the main function. main is also a method/function

```
void main()
{
    // body part
}
```

Method Definition

- A method definition defines the method header and body.
- A method body part defines method logic.

Syntax:	return-type method_name(datatyp1 arg1, datatype2 arg2, …) { functions statements }
Example:	int addition(int a, int b); { return a+b; }

WAP to add two number using add(int, int) method

```
1. class MethodDemo{
2.  public static void main(String[] args) {
3.   int a=10,b=20,c;
4.   MethodDemo md=new MethodDemo();
5.   c=md.add(a,b);
6.   System.out.println("a+b="+c);
7.  }//main()

8.   int add(int i, int j){
9.                          return i+j;
10.                       }
11.}
```

Output:

```
a+b=30
```

Actual parameters v/s Formal parameters

- Values that are passed from the calling functions are known actual parameters.
- The variables declared in the function prototype or definition are known as formal parameters.
- Name of formal parameters can be same or different from actual parameters.
- Sequence of parameter is important, not name.

Actual Parameters

```
int a=10,b=20,c;
MethodDemo md=new MethodDemo();
c=md.add(a,b);
// a and b are the actual parameters.
```

Formal Parameters

```
int add(int i, int j)
{
            return i+j;
}
// i and j are the formal parameters.
```

Return Statement

- The function can return only one value.
- Function cannot return more than one value.
- If function is not returning any value then return type should be void.

Main Program

```
int a=10,b=20,c;
MethodDemo md=new MethodDemo();
c=md.sub(a,b);
// a and b are the actual parameters.
```

Method

```
int sub(int i, int j)
{
      return i - j;
}
// i and j are the formal parameters.
```

WAP to calculate the Power of a Number using method

```
1. class MethodDemo{
2. import java.util.*;
3. public class PowerMethDemo1{
4. public static void main(String[] args){
5.   int num, pow, res;
6.   Scanner sc=new Scanner(System.in);
7.   System.out.print("enter num:");
8.   num=sc.nextInt();
9.   System.out.print("enter pow:");
10.       pow=sc.nextInt();
11.       PowerMethDemo1 pmd=new PowerMethDemo1();
12.       res = pmd.power(num, pow);
13.       System.out.print("ans="+res);
14.       } //main()
15.int power(int a, int b){
16.    int i, r = 1;
17.    for(i=1; i<=b; i++)
18.    {
19.           r = r * a;
20.    }
21.    return r;
22.  }//power()
23.}//class
```

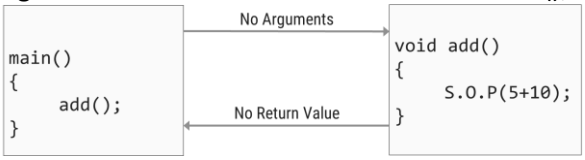
Output:

```
enter num:5
enter pow:3
ans=125
```

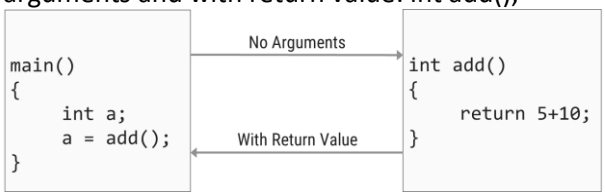
Types of Methods(Method Categories)

Functions can be divided in 4 categories based on arguments and return value.

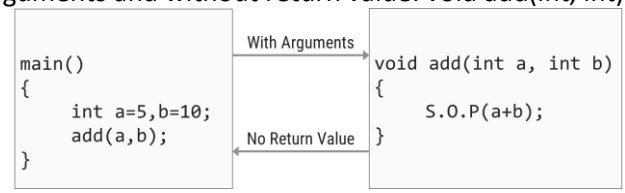
- Method without arguments and without return value: void add();



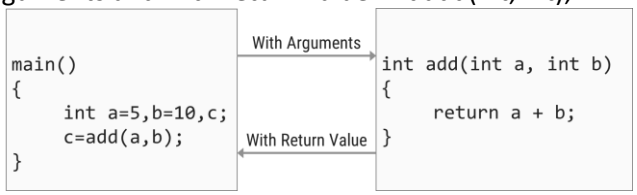
- Method without arguments and with return value: int add();



- Method with arguments and without return value: void add(int, int);



- Method with arguments and with return value: int add(int, int);



Advantages of Method

Reduced Code Redundancy

- Rewriting the same logic or code again and again in a program can be avoided.

Reusability of Code

- Same function can be call from multiple times without rewriting code.

Reduction in size of program

- Instead of writing many lines, just function need to be called.

Saves Development Time

- Instead of changing code multiple times, code in a function need to be changed.

More Traceability of Code

- Large program can be easily understood or traced when it is divide into functions.

Easy to Test & Debug

- Testing and debugging of code for errors can be done easily in individual function.

Method Overloading

- **Definition**: When two or more methods are implemented that share same name but different parameter(s), the methods are said to be overloaded, and the process is referred to as method overloading.
- Method overloading is one of the ways that Java implements **polymorphism**.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- E.g. **public void draw()**
  public void draw(int height, int width)
  public void draw(int radius)
- Thus, overloaded methods must differ in the type and/or number of their parameters.
- While in overloaded methods with different return types and same name & parameter are not allowed, as the return type alone is insufficient for the compiler to distinguish two versions of a method.

Method Overloading: Compile-time Polymorphism

```
1. class Addition{
2. int i,j,k;
3.   void add(int a){
4.       i=a;
5.       System.out.println("add i="+i);
6.   }
7.   void add(int a,int b){\\overloaded add()
8.       i=a;
9.       j=b;
10.           System.out.println("add i+j="+(i+j));
11.  }
12.      void add(int a,int b,int c){\\overloaded add()
13.           i=a;
14.           j=b;
15.           k=c;
16.           System.out.println("add i+j+k="+(i+j+k));
17.       }
18.      }
19.      class OverloadDemo{
20.      public static void                  main(String[] args){
21.       Addition a1= new Addition();
22.       //call all versions of add()
23.           a1.add(20);
24.           a1.add(30,50);
25.           a1.add(10,30,60);
26.           }
27.      }
```

Output:

```
add i=20
add i+j=80
add i+j+k=100
```

Method Overloading :Points to remember

- Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm.
- Overloading increases the readability of the program.
- There are two ways to overload the method in java
    - By changing number of arguments
    - By changing the data type
- In java, method overloading is not possible by changing the return type of the method only because of ambiguity.

Can we overload java main() method?

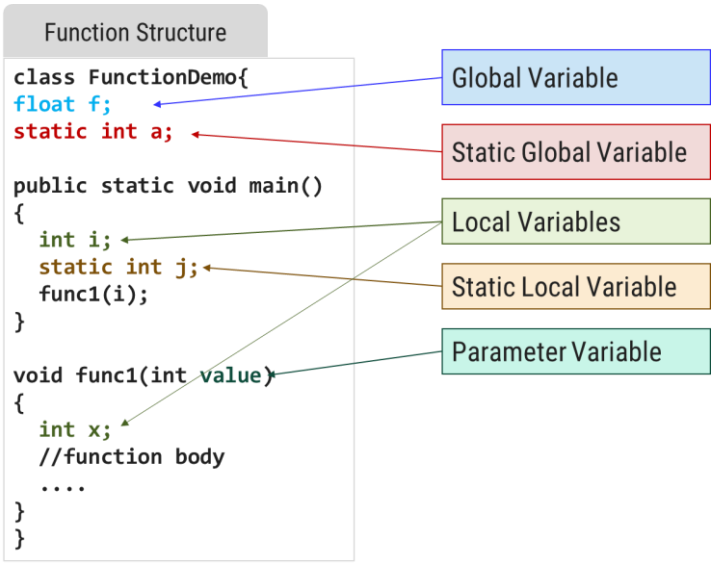
- Yes, by method overloading. We can have any number of main methods in a class by method overloading
- But JVM calls main () method which receives string array as arguments only.

Scope, Lifetime and Visibility of a Variable

Whenever we declare a variable, we also determine its scope, lifetime and visibility.

Scope	Scope is defined as the area in which the declared variable is 'accessible'. **There are five scopes**: program, file, function, block, and class. **Scope** is the region or section of code where a variable can be accessed. Scoping has to do with **when** a variable is accessible and used.
Lifetime	The lifetime of a variable is the period of time in which the variable is allocated a space (i.e., the period of time for which it "lives"). There are three lifetimes in C: static, automatic and dynamic. Lifetime is the time duration where an object/variable is in a valid state. Lifetime has to do with when a variable is created and destroyed
Visibility	Visibility is the "accessibility" of the variable declared. It is the result of hiding a variable in outer scopes.

Scope of a Variable



Scope	*Description*
Local (block/function)	"visible" within function or statement block from point of declaration until the end of the block.
Class	"seen" by class members.
File(program)	visible within current file.
Global	visible everywhere unless "hidden".

Lifetime of a variable

- The lifetime of a variable or object is the time period in which the variable/object has valid memory.
- Lifetime is also called "allocation method" or "storage duration".

	Lifetime	***Stored***
Static	Entire duration of the program's execution.	data segment
Automatic	Begins when program execution enters the function or statement block and ends when execution leaves the block.	function call stack
Dynamic	Begins when memory is allocated for the object (e.g., by a call to malloc() or using new) and ends when memory is deallocated (e.g., by a call to free() or using delete).	heap

Variable Type	***Scope of a Variable***	***Lifetime of a Variable***
Instance Variable	Throughout the class except in static methods	Until object is available in the memory
Class Variable	Throughout the class	Until end of the Class
Local Variable	Throughout the block/function in which it is declared	Until control leaves the block

Introduction to Classes in java

What is Class?

- Class is derived datatype, it combines members of different datatypes into one.
- Defines new datatype (primitive ones are not enough).
- For Example: Car, College, Bus etc.
- This new datatype can be used to create objects.
- A class is a template for an object.

```
class Car{
        String company;
        String model;
        double price;
        double mileage;
        ……
}
```

Class: Car

Properties (Describe)
Company
Model
Color
Mfg. Year
Price
Fuel Type
Mileage
Gear Type
Power Steering
Anti-Lock braking system

Methods (Functions)
Start
Drive
Park
On_break
On_lock
On_turn

Introduction to Objects in java

What is Object?

- An object is an instance of a class.
- An object has a state and behavior.
  Example: A dog has
  states - color, name, breed as well as
  behaviors – barking, eating.
- The state of an object is stored in fields (variables), while methods (functions) display the object's behavior.
- An Object is a key to understand Object Oriented Technology.
- An entity that has state and behavior is known as an object. e.g., Mobile, Car, Door, Laptop etc

- Each and every object posses
    1. Identity
    2. State
    3. Behavior

Philosophy of Object Oriented

- Our real world is nothing but classification of objects
    - E.g. Human, Vehicle, Library, River, Watch, Fan, etc.
- Real world is organization of different objects which have their own characteristics, behavior
    - Characteristic of Human: Gender, Age, Height, Weight, Complexion, etc.
    - Behavior of Human: Walk, Eat, Work, React, etc.
    - Characteristic of Library: Books, Members, etc.
    - Behavior of Library: New Member, Issue Book, Return Book etc.
- The OO philosophy suggests that the things manipulated by the program should correspond to things in the real world.
    - Classification is called a Class in OOP
    - Real world entity is called an Object in OOP
    - Characteristic is called Property in OOP
    - Behavior is called Method in OOP


Pen


Board


Laptop

Result

Bank Account


Bench

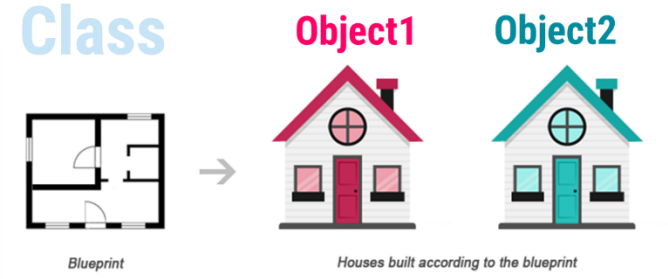

Projector


Bike

Physical objects...

Logical objects...

Classes and Objects



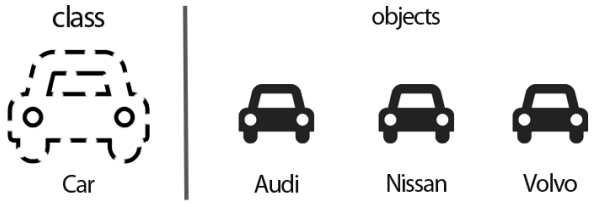
Class is a blueprint of an object
Class describes the object

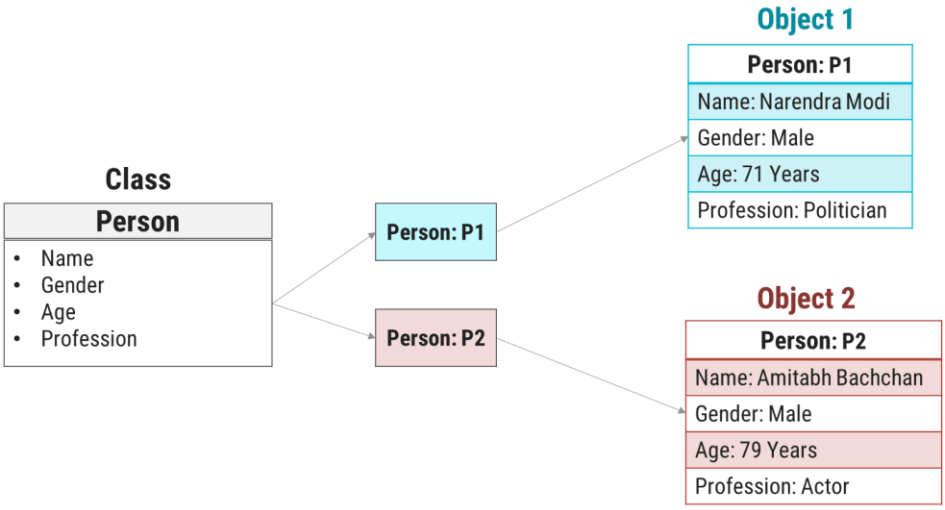
Object is instance of class

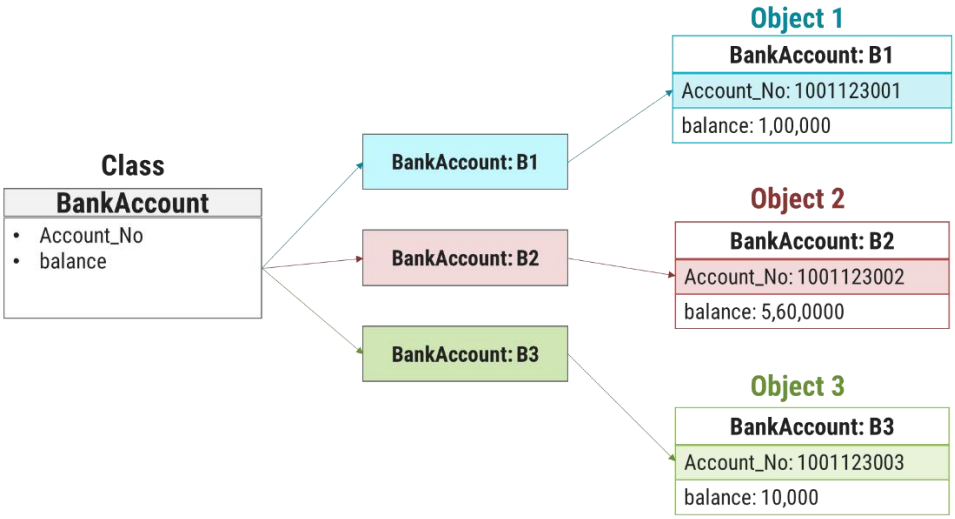
- Class can be defined in multiple ways
  - A class is the building block.
  - A class is a blueprint for an object.
  - A class is a user-defined data type.
  - A class is a collection of objects of the similar kind.
  - A class is a user-defined data type which combines data and methods.
  - A class describes both the data and behaviors of objects.
- Class contains data members (also known as field or property or data) and member functions (also known as method or action or behavior)
- Classes are similar to structures in C.
- Class name can be given as per the Identifier Naming Conventions.

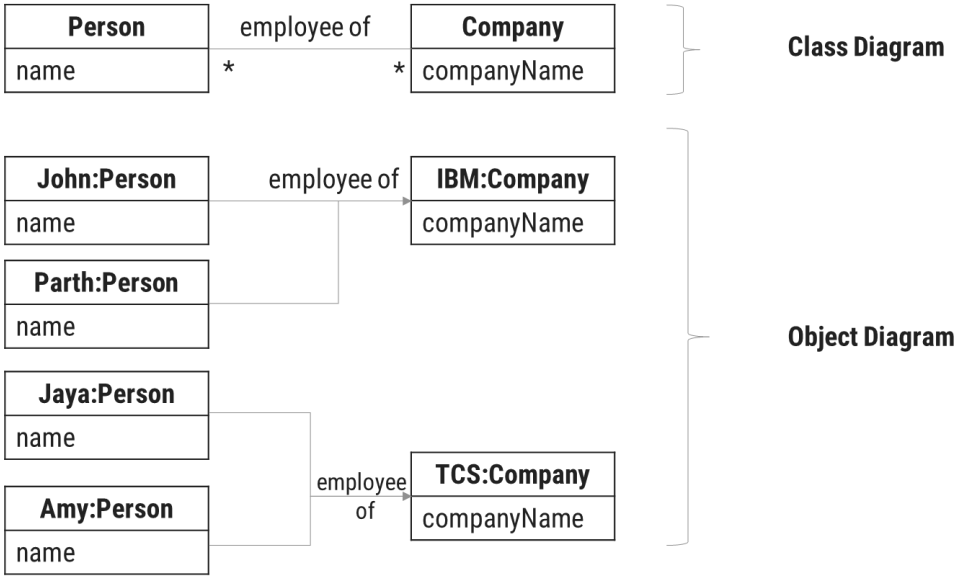
Object Definition:

- An Object is an instance of a Class.
- An Object is a variable of a specific Class
- An Object is a data structure that encapsulates data and functions in a single construct.
- Object is a basic run-time entity
- Objects are analogous to the real-world entities.









Creating Object & Accessing members

- new keyword creates new object
- Syntax:

  ClassName objName = new ClassName();
- Example :

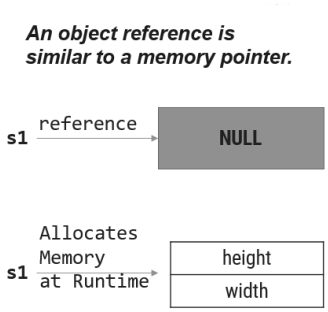
  SmartPhone iPhone = new SmartPhone();
- Object variables and methods can be accessed using the dot (.) operator
- Example: iPhone.storage = 8000;

Declaring an Object

- When we create a class, we are creating a new data type.
- Object of that data type will have all the attributes and abilities that are designed in the class.
- The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.
- This reference is, more or less, the address in memory of the object allocated by new.

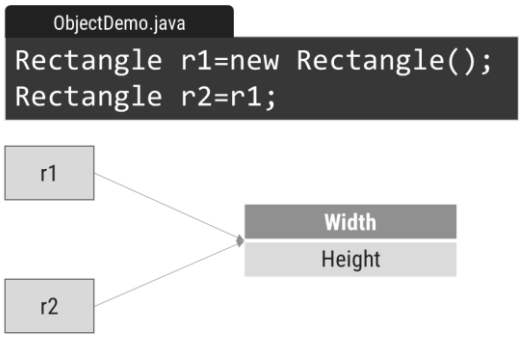
- This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

```
1. class Square{
2.      double height;
3.      double width;
4. }
5. class MyProg{
6.      public static void main(String[] args) {
7.         Square s1; //declare reference to object
8.         s1= new Square();//allocate a Square object
9.   }
10.}
```

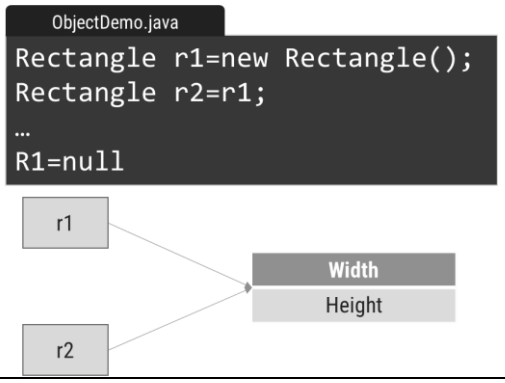


- ***new*** operator dynamically allocates memory for an object
- Here, s1 is a variable of the class type.
- The class name followed by parentheses specifies the constructor for the class.
- It is important to understand that new allocates memory for an object during run time.

Assigning Object Reference



Here, r1 and r2 will both refer to the same object. The assignment of r1 to r2 did not allocate any memory or copy any part of the original object. It simply makes r2 refer to the same object as does r1



Here, r1 has been set to null, but r2 still points to the original object.

WAP using class Person to display name and age

```
1. class MyProgram {
2. public static void main(String[] args) {
3.   Person p1= new Person();
4.   Person p2= new Person();
5.   p1.name="modi";
6.   p1.age=71;
7.   p2.name="bachchan";
8.   p2.age=80;
9.   System.out.println("p1.name="+p1.name);
10.      System.out.println("p2.name="+p2.name);
11.        System.out.println("p1.age="+p1.age);
12.        System.out.println("p2.age="+p2.age);
13.       }//main()
14.      }//class myProgram

15.      class Person
16.      {
17.            String name;
18.            int age;
19.      }//class person
20.
Output:
p1.name=modi
p2.name=bachchan
p1.age=71
p2.age=80
```

WAP using class Person to display name and age with method

```
1. class MyProgram {
2. public static void main(String[] args){
3.   Person p1=new Person();
4.   Person p2=new Person();
5.   p1.name="modi";
6.   p1.age=71;
7.   p2.name="bachchan";
8.   p2.age=80;
9.   p1.displayName();
10.        p2.displayName();
11.        p1.displayAge();
12.        p2.displayAge();
13.       } //main()
14.      } //class myProgram

15. class Person{
16.      String name;
17.      int age;
18.   public void displayName(){
19.      System.out.println("name="+name);
20.   }
21.   public void displayAge(){
22.      System.out.println("age="+age);
23.   }
24. }//class person
```

Output:

```
name=modi
name=bachchan
age=71
age=80
```

WAP using class Rectangle and calculate area using method

```
1. import java.util.*;
2. class MyProgram {
3.   public static void main(String[] args){
4.   Rectangle r1=new Rectangle();
5.   Scanner sc=new Scanner(System.in);
6.   System.out.print("enter height:");
7.   r1.height=sc.nextFloat();
8.   System.out.print("enter width:");
9.   r1.width=sc.nextFloat();
10.        r1.calArea();
11.       } //main()
12.      }//class myProgram

13. class Rectangle{
14.      float height;
15.      float width;
16.      public void calArea() {
17.      System.out.println( "Area="+height*width);
18.       } //calArea()
19.     } //class
```

```
Output:
enter height:30.55
enter width:20.44
Area=624.442
```

WAP using class Rectangle and calculate area with Return value

```
1. import java.util.*;
2. class MyProgram {
3. public static void main(String[] args){
4.   float area;
5.   Rectangle r1=new Rectangle();
6.   Scanner sc=new Scanner(System.in);
7.   System.out.print("enter height:");
8.   r1.height=sc.nextFloat();
9.   System.out.print("enter width:");
10.        r1.width=sc.nextFloat();
11.        area=r1.calArea();
12.        System.out.println("Area="+area);
13.       }//main()
14.      }//class myProgram

15. class Rectangle{
16.      float height;
17.      float width;
18.      public float calArea() {
19.            return height*width;
20.       }//calArea()
21.      }//class
```

Output:

```
enter height:30.55
enter width:20.44
Area=624.442
```

WAP using class Cube and calculate area using method with parameter

```
1. import java.util.*;
2. class MyProgramCube {
3.  public static void main
4.                   (String[] args){
5.   float area;
6.   Cube c1= new Cube();
7.   area=c1.calArea(10,10,10);
8.   System.out.println("area="+area);
9.    }//main()
10.       }//class myProgram
11.      class Cube{
12.      float height;
13.      float width;
14.      float depth;
15.      float calArea(float h, float w, float d)
16.      {   height=h;
17.          width=w;
18.          depth=d;
19.            return height*width*depth;
20.       }//calArea()
21.      }//class
```

Output:

```
area=1000.0
```

WAP using class Cube and calculate area of two objects

```
1. import java.util.*;
2. class MyProgramCube {
3. public static void main(String[] args){
4.       float area;
5.       Cube c1= new Cube(); //Obj1
6.       Cube c2= new Cube(); //Obj2
7.       System.out.println("c1 area="+c1.calArea(10,10,10));
8.       System.out.println("c2 area="+c2.calArea(20,20,20));
9.       } //main()
10.      } //class

11.  class Cube{
12.      float height;
13.      float width;
14.      float depth;
15.      float calArea(float h, float w, float d)
16.      {   height=h;
17.          width=w;
18.          depth=d;
19.            return height*width*depth;
20.       } //calArea()
21.      } //class
22.
Output:
c1 area=1000.0
c2 area=8000.0
```

Class vs. Object

Class	Object
Class is a Blueprint or template.	Object is the instance of a class.
Class creates a logical framework that defines the relationship between its members.	When you declare an object of a class, you are creating an instance of that class.
Class is a logical construct.	An object has physical reality. (i.e., an object occupies space in memory.)
Class is a group or collection of similar object. e.g. Car	An Object is defined as real-world entity e.g.: Audi, Volkswagen, Tesla, Ferrari etc.
Class is declared only once	An Object can be created as many times as required
Class doesn't allocate memory when it is created.	An Object allocates the memory upon creation
Class can exist without its objects.	An Object can't exist without a class.
Class: Profession **Class**: Mobile **Class**: Subject **Class**: Student **Class**: Color	**Object**: Doctor, Teacher, Lawyer, Politician… **Object**: iPhone, Samsung, 1plus… **Object**: Maths, English, Science, Computer… **Object**: John, Aarav, Smita… **Object:** Blue, Green, Red, Yellow, Violet, Black…

Constructor

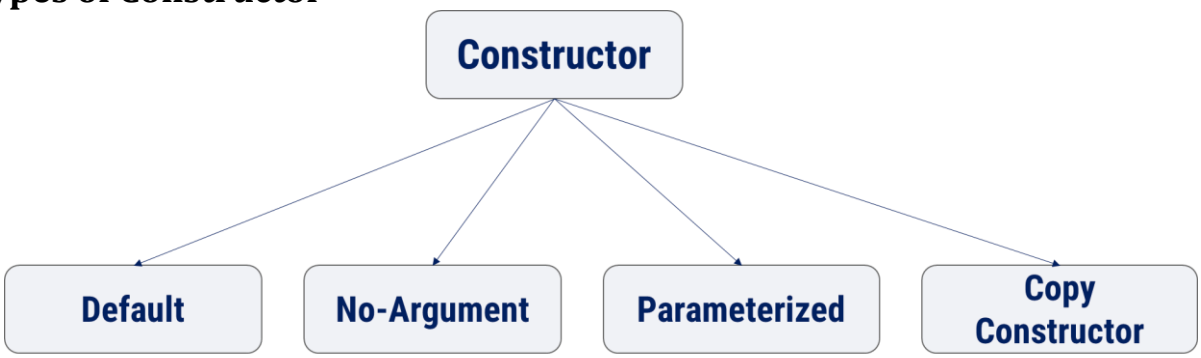
- A constructor in Java is a special type of method that is used to initialize objects.
- The constructor is called when an object of a class is created.
- A constructor initializes an object immediately upon creation.
- It has the same name as the class in which it resides and is syntactically similar to a method.
- JVM first allocates the memory for variables (objects) and then executes the constructor to initialize instance variables.
- The JVM calls it automatically when we create an object.
- A constructor defines what happens when an object of a class is created.

```
class Cube{
instance variable1
instance variable1
…
   Cube()
   {
              //initailze object

   }
} //class
```

Properties of Constructor

- Constructor is invoked automatically whenever an object of class is created.
- Constructors do not have return types and they cannot return values, not even void.
- All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you known as default constructor.
- Constructor is a method that is called at runtime during the object creation by using the new operator. The JVM calls it automatically when we create an object.
- It is called and executed only once per object.
- It means that when an object of a class is created, constructor is called. When we create 2nd object then the constructor is again called during the second time.

Types of Constructor



Default Constructor

- A constructor defines what occurs when an object of a class is created.
- Most classes explicitly define their own constructors within their class definition but if no explicit constructor is specified then java will automatically supply a default constructor.
- Once you define your own constructor, the default constructor is no longer used.
- The default constructor automatically initializes all instance variables to zero.

Code before compilation:

```
class MyConst {
public static void main(String[] args) {
         MyConst c=new MyConst();
   }
}
```

Code after compilation:

```
class MyConst{
 MyConst(){
        //Default Constructor…
        }
public static void main(String[] args)   {
      c=new MyConst();
   }
}
```

No-Argument Constructor: MyMain.java

```
1. class Cube {
2.       double width;
3.       double height;
4.       double depth;
5.       Cube()
6.       {
7.             System.out.println("Constructing cube");
8.             width = 10;
9.             height = 10;
10.            depth = 10;
11.        }//Cube()
12.      }//class
13.  class MyMain{
14.  public static void main(String[] args){
15.             Cube c=new Cube();
16.      }
17.  }
```

Note: *If you implement any constructor then you no longer receive a default constructor from Java compiler.*

Parameterized Constructor

Parameterized Constructor: MyMain.java

```
1. class Cube {
2.       double width, height, depth;
3.       Cube(double w, double h, double d)
4.       {     System.out.println("Constructing cube");
5.             width = w;
6.             height = h;
7.             depth = d;
8.       }//Cube()
9. }//class
10.    class MyMain{
11.    public static void main(String[] args){
12.      Cube c=new Cube(10,10,10);
13.      }
14.    }
```

Copy Constructor

- It is a special type of constructor that is used to create a new object using the existing object of a class that had been created previously.

Copy Constructor: MyProgramCopy.java

```
1. class Student{
2.       String name;
3.       int rollno;
4.       Student(String s_name, int s_roll){
5.             System.out.println("ConstructorInvoked");
6.              this.name=s_name;
7.             this.rollno=s_roll;
8.       } //Constructor1

9.       Student(Student s){ //CopyConstructor
10.            System.out.println("CopyConstructor Invoked");
11.            this.name=s.name;
12.            this.rollno=s.rollno;
13.         } //Constructor2

14.         public void display(){
15.          System.out.print("name="+name);
16.          System.out.println("  rollno="+rollno);
17.       } // display()
18.      } //class

19.      class MyProgramCopy {
20.      public static void main(String[] args){
21.        Student s1=new Student("darshan",101);
22.        //invoking Copy Constructor
23.        Student s2=new Student(s1);
24.        s1.display();
25.        s2.display();
26.       } //main()
27.      } //class myProgram
Output:
Constructor Invoked
CopyConstructor Invoked
name=darshan  rollno=101
name=darshan  rollno=101
```

- It creates a new object by initializing the object with the instance of the same class.

Advantages of Copy Constructor

- It is easier to use when our class contains a complex object with various parameters.
- Whenever we need to add all the field of a class to another object, then just send the reference of previously created object.
- One of the most importance of copy constructors is that there is no need for any typecasting.
- Using a copy constructor, we can have complete control over object creation.
- With Copy Constructor, we can pass object of the class as a parameter(pass by reference).

Why Constructor?

- The pivotal purpose of constructor is to initialize the instance variable of the class.
- We use constructors to initialize the object with the default or initial state.
- Through constructor, we can request the user of that class for required dependencies.
- A constructor within a class allows constructing the object of the class at runtime.
- Allocates appropriate memory to objects.
- If we need to execute some code at the time of object creation, we can write them inside the constructor.
- Example:
    - If we talk about a Cube class then class variables are width, height and depth.
    - But when it comes to creating its object(i.e. Cube will now exist in the computer's memory), then can a cube be there with no value defined for its dimensions? The answer is "NO".
    - So constructors are used to assign values to the class variables at the time of object creation.

When to use Constructor?

- When we need to execute some code at the time of object creation.
- Used for the initialization of instance variables.
- To assign the default value to instance variables.
- To initializing objects of the class.

Constructor() vs. Method()

	Constructor()	Method()
Naming	Constructor name must be same as class name.	Method name can be anything.
Return types	Constructor does not have any return type, not even void.	Method must have return types, at least void.
Call	Constructor can be invoked implicitly when object is created.	Method is called by the programmer. Invoked explicitly.
Purpose	To initialize an object	To execute the code
Inheritance	Constructor cannot be inherited by subclass.	Method can be inherited by subclass.

Constructor Overloading

- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- They are arranged in a way that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters in the list and their types.

Constructor Overloading

```
1. class Balance{
2.       int accNo;
3.       double bal;
4.       Balance(){
5.         System.out.println("inside const1");
6.         bal=0;
7.       }
8.       Balance(double b){
9.         System.out.println("inside const2");
10.           bal=b;
11.            }
12.      Balance(int a,double b){
13.              System.out.println("inside const3");
14.              bal=b;
15.              accNo=a;
16.            }
17.      }
18.  class Account{
19.      public static void main(String  args[]){
20.            Balance b1= new Balance();
21.            Balance b2= new Balance(100);
22.            Balance b3=new Balance(1201,10000);
23.            System.out.println("b1.bal="+b1.bal);
24.            System.out.println("b2.bal="+b2.bal);
25.            System.out.println("b3.bal="+b3.bal+
                                          "b3.accNo="+b3.accNo);
26.            }
27.      }
```

Destructor

- Destructor is the opposite of the constructor. Constructor is used to initialize objects while the destructor is used to delete or destroy the object that releases the resource occupied by the object.
- Definition: Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed.
- In other words, a destructor is the last function that is going to be called before an object is destroyed.
- In java, there is a special method named garbage collector that automatically called when an object is no longer used.
- When an object completes its life-cycle the garbage collector deletes that object and de-allocates or releases the memory occupied by the object.

- In C++, dynamically allocated objects must be manually released by use of a delete operator.
- Java takes a different approach: it handles de-allocation automatically.
- The technique that accomplishes this is known as “garbage collection”.

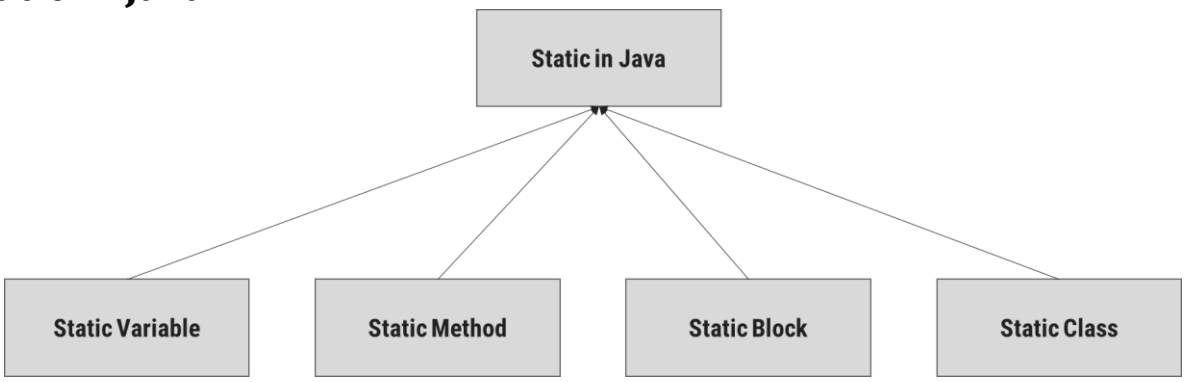
Why Destructor?

- When we create an object of the class(using new), it occupies some space in the memory. If we do not delete these objects, it remains in the memory and occupies unnecessary space.
- To resolve this problem, we use the destructor.
- ***Remember that there is no concept of destructor in Java.***
- Instead of destructor, Java provides the garbage collector that works the same as the destructor.
- The garbage collector is a program (thread) that runs on the JVM. It automatically deletes the unused objects (objects that are no longer used) and free-up the memory.
- The programmer has no need to manage memory, manually.

Working of garbage collector(destructor) in java

- When the object is created it occupies the space in the heap. These objects are used by the threads.
- If the objects are no longer used by the thread it becomes eligible for the garbage collection.
- The memory occupied by that object is now available for new objects that are being created.
- When the garbage collector destroys an object, the JRE calls finalize() method to close the connections such as database and network connection.

Static in Java



- The static keyword is used for memory management.
- We can apply static keyword with variables, methods, blocks and nested classes.
- The static keyword belongs to the class than an instance of the class.
- The static can be:
    1. Variable (also known as a class variable)
    2. Method (also known as a class method)
    3. Block
    4. Nested class

Static Variable

- Static variables have a property of preserving their value even after they are out of their scope.
- The static variable gets memory only once in the class area at the time of class loading.
- **Advantage of static variable**: It makes program memory efficient (static variable saves memory).
- **Syntax**: `static` *type variable_name*;
-

Characteristics of static variable:

- It is initialized to zero when the first object of its class is created. No other initialization is allowed.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.
- Static variables are normally used to maintain values common for all objects.
- The class constructor does not initialize static variable.

Static Method vs Non-Static Method

MyProgram.java(Non-static function)

```
1. public class MyProgram {
2. public static void
       main(String[] args) {
3.   int a=1,b=2,c;
4.   MyProgram mp=new
                  MyProgram();
5.   c = mp.add(a,b);
6.   System.out.println(c);
   }//main
7.  public int add(int i,int j)
8.  {
9.      return i + j;
10. }
11.}//class
```

MyProgram.java(static function)

```
1. public class MyProgram {
2. public static void
       main(String[] args) {
3. int a=1,b=2,c;
4. c = add(a,b);
5. System.out.println(c);
6. }//main


7. static int add(int i,int j)
8. {          return i + j;
9. }
10.}//class
```

-

Characteristic of static method

- A static method can call only other static methods and cannot call a non-static method from it.
- A static method can be accessed directly by the class name and doesn't need any object
- A static method cannot refer to "this" or "super" keywords in anyway

Restrictions on static method

1. The static method cannot use non static data member or call non-static method directly.
2. **this** and **super** cannot be used in static context.

Static Method: WAP using class Rectangle and calculate area

```
1. import java.util.*;
2. class Rectangle{
3.       static float height;
4.       static float width;
5.       static void calArea() {
6.             System.out.println( "Area= "+height*width);
7.       } //calArea()
8. } //class
9. class MyRectangle {
10.      public static void main(String[] args){
11.        Rectangle r1= new Rectangle();
12.        Scanner sc= new Scanner(System.in);
13.        System.out.print("enter height:");
14.        r1.height=sc.nextFloat();
15.        System.out.print("enter width:");
16.        r1.width=sc.nextFloat();
17.        Rectangle.calArea();
18.       } //main()
19.      } //class
```

Output:

```
enter height:30.55
enter width:20.44
Area=624.442
```

Static Block

- Static block is executed exactly once, when the class is first loaded.
- It is used to initialize static variables of the class.
- It will be executed even before the main() method.

```
static {
        //initialisation of static variables…
      }
```

How to call static block in java?

- Unlike method, there is no specified way to call a static block.
- The static block executes automatically when the class is loaded in memory.

Example: Static Block, Method and Variable

```
1. class StaticDemo {
2.       static int a = 4; //static variable declared & initialized
3.       static int b;     //static variable declared
4.       static void dispValue(int x) {
5.       System.out.println("Static method initialized.");
6.       System.out.println("x = " + x);
7.       System.out.println("a = " + a);
8.       System.out.println("b = " + b);
9.       } //static method

10.      static {
11.                  System.out.println("Static block initialized.");
12.                   b= a * 5;
13.            } //static block
14.      public static void main(String args[]) {
15.            System.out.println("inside main()...");
16.            dispValue(44);
17.            } //main()
18.     } //class
```

Output:

```
Static block initialized.
inside main()...
Static method initialized.
x = 44
a = 4
b = 20
```

Points to remember for static keyword

1. When we declare a field static, exactly a single copy of that field is created and shared among all instances of that class.
2. Static variables belong to a class, we can access them directly using class name. Thus, we don't need any object reference.
3. We can only declare static variables at the class level.
4. We can access static fields without object initialization.
5. Static methods can't be overridden.
6. Abstract methods can't be static.
7. Static methods can't use this or super keywords.
8. Static methods can't access instance variables and instance methods directly. They need some object reference to do so.
9. A class can have multiple static blocks.

Mutable and Immutable Objects

The content of mutable object can be changed, while content of immutable objects cannot be changed.

```java
class MutableClass{
    int a;
    void add5() {
        a = a + 5;
    }
}
public class MutableClassDemo {
    public static void main(String[] args) {
        MutableClass m1 = new MutableClass();
        m1.a = 10;
        m1.add5();
        System.out.println(m1.a);
    }
}
```

```java
class ImmutableClass{
    int a;
    int add5() {
        return ( a + 5 );
    }
}
public class MutableClassDemo {
    public static void main(String[] args) {
        ImmutableClass m1 = new ImmutableClass();
        m1.a = 10;
        int ans = m1.add5();
        System.out.println("A in m1 = " + m1.a);
        System.out.println("returned = " + ans);
    }
}
```

Passing Objects as Argument

In order to understand how and why we need to pass object as an argument in methods, lets see the

Example: Passing Objects as Argument

```
1. class Time{
2.       int hour;
3.       int minute;
4.       int second;
5.       public Time(int hour, int minute, int second) {
6.             this.second = second;
7.             this.minute = minute;
8.             this.hour = hour;
9.       }

10.      void add(Time t) {
11.            this.second += t.second;
12.            if(this.second>=60) {
13.                  this.minute++;
14.                  this.second-=60;
15.      }
16.      this.minute += t.minute;
17.      if(this.minute>=60) {
18.                  this.hour++;
19.                  this.minute-=60;
20.      }//if
21.      this.hour += t.hour;
22.      }
23.      }
24.   public class TimeDemo {
25.      public static void main(String[] args) {
26.      Time t1 = new Time(11,59,55);
27.      Time t2 = new Time(0,0,5);
28.      t1.add(t2);//passing Object as an argument
29.      System.out.println(t1.hour + ":" + t1.minute + ":" +
                                                      t1.second);
30.      }
31.    }
32.
```

example.

Array of Objects

- We can create an array of object in java.
- Similar to primitive data type array we can also create and use arrays of derived data types

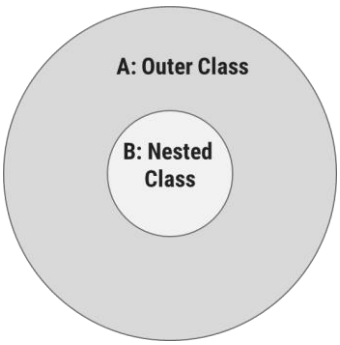
Example: Array of Objects

```
1. class Student{
2.     int rollNo;
3.     String name;
4.     public Student(int rollNo, String name) {
5.                                 this.rollNo = rollNo;
6.                                 this.name = name;
7.     }
8.     void printStudentDetail() {
9.             System.out.println("| "+rollNo+" | -- | "+name +" |");
10.     }
11.    }
12.  public class ArrayOfObjectDemo {
13.    public static void main(String[] args) {
14.    Student[] stu = new Student[3];
15.    stu[0] = new Student(101,"darshan");
16.    stu[1] = new Student(102, "OOP");
17.    stu[2] = new Student(103,"java");
18.    stu[0].printStudentDetail();
19.    stu[1].printStudentDetail();
20.    stu[2].printStudentDetail();
21.    }
22.   }
```

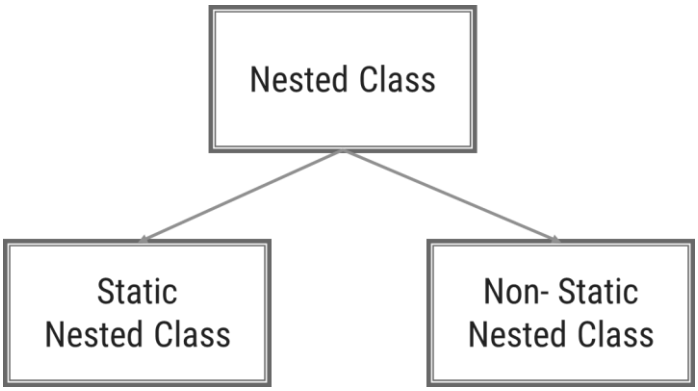
(class).

Nested Class

- Nested Class: Class within another class
- Scope: Nested class is bounded by the scope of its enclosing class.
    - E.g. class B is defined within class A, then B is known to A, but not outside of A.
- A nested class has access to the members, including private members of the class in which it is nested.
- However, the enclosing class does not have access to the members of the nested class. i.e. Class B can access private member of class A, while reverse is not accessible.



Types of Nested class:



Non-Static Nested Class

- The most imp type of nested class is the inner class.
- An inner class is a non-static nested class.
- It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static member of the outer class do.

Non-Static Nested Class: InnerOuterDemo.java

```
1. class Outer{
2.    private int a=100;//instance variable
3.    void outerMeth(){
4.      Inner i= new Inner();
5.      System.out.println("inside outerMeth()...");
6.      i.innerMeth();
7.    }
8.    class Inner{
9.      int b=20;
10.        void innerMeth(){
11.            System.out.println("inside innerMeth()..."
12.                                        +(a+b));
13.            }
14.       } //inner class
15.   } //outer class

16.   class InnerOuterDemo{
17.   public static void main(String[] args)
18.      {
19.            Outer o= new Outer();
20.            o.outerMeth();
21.       }
22.   } //InnerOuterDemo
```

Output:

```
inside outerMeth()...
inside innerMeth()...120
```

Static Nested Class: InnerOuterDemo

- A static nested class is one which has the static modifier applied because it is static, it must access the member of its enclosing class through an object. i.e. it can not refer to members of its enclosing class directly.
- Because of this reason, static nested classes are rarely used.

Static Nested Class: InnerOuterDemo

```
1. class Outer{
2.
3.    void outerMeth(){
4.      Inner i= new Inner();
5.      System.out.println("inside outerMeth()...");
6.      i.innerMeth();
7.    }
8.    static class Inner{
9.      int b=20;
10.        void innerMeth(){
11.             System.out.println("inside innerMeth()..."
12.                                        +(a+b));
13.           }
14.       } //inner class
15.   } //outer
16.   class InnerOuterDemo{
17.   public static void main(String[]                    args)
18.      {
19.            Outer o= new Outer();
20.            o.outerMeth();
21.       }
22.   } //InnerOuterDemo
```

Output:

```
inside outerMeth()...
inside innerMeth()...120
```

Points to remember: Inner class

- Inner class implements a security mechanism in Java.
- Reduces encapsulation, more organized code by logically grouping the classes.

import keyword

- import keyword is used to import built-in and user-defined packages into your java source file so that your class can refer to a class that is in another package by directly using its name.
- There are 3 different ways to refer to class/interface that is present in different package
    - import the class/interface you want to use.
    - import all the classses/interfaces from the package.
    - Using fully qualified name.
- We can import a class/interface of other package using a import keyword at the first line of code.

```
import java.util.Scanner;
public class DemoImport {
	public static void main(String[] args)
	{
		Scanner s = new Scanner(System.in);
		// Code
	}
}
```

- We can import all the classes/interfaces of other package using a import keyword at the first line of code with the wildcard (*).

```
import java.util.*;
public class DemoImport {
	public static void main(String[] args)
	{
		Scanner s = new Scanner(System.in);
		Date d = new Date();
		// Code
	}
}
```

- It is possible to use classes from other packages without importing the class using fully qualified name of the class.
- Example :

```
java.util.Scanner s = new java.util.Scanner(System.in);
```

Static Import

- The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly.
- **Advantage**: Less coding is required if you have to access any static member of a class more frequently.
- **Disadvantage**: If you overuse the static import feature, it makes the program unreadable and unmaintainable.

```
import static java.lang.System.out;
public class S2{
  public static void main(String args[]){
     out.println("Hello main");
  }
}
```

Access Control

Modifier	Same Class	Same Package Sub Class	Same Package Non Sub Class	Different Package Sub Class	Different Package Non Sub Class
Private	☑				
Default	☑	☑	☑		
Protected	☑	☑	☑	☑	
Public	☑	☑	☑	☑	☑

What is OOP?

- OOP (Object-Oriented Programming) is a programming paradigm that is completely based on 'objects'.
- Object-Oriented Programs insists to have a lengthy and extensive design phase, which results in improved designs and fewer defects.
- Object-oriented programming provides a higher level way for programmers to envision and develop their applications.
- In an object-oriented programming language, less emphasis is placed upon the flow of execution control. Instead, the program is viewed as a set of objects interacting with each other in defined ways.
- An OOP programmer can bind new software objects to make completely new programs/system.

What is Object-Orientation?

- Object-oriented modeling and design is a way of thinking about problems using models organized around real-world concepts.
- The fundamental construct is object, which combine data structure and behavior.
- Object-Oriented Models are useful for
    - Understanding problems
    - Communicating with application experts
    - Modeling enterprises
    - Preparing documentation
    - Designing System

Thinking in objects and class relationships

Few Real World Classes with behavior / action / methods

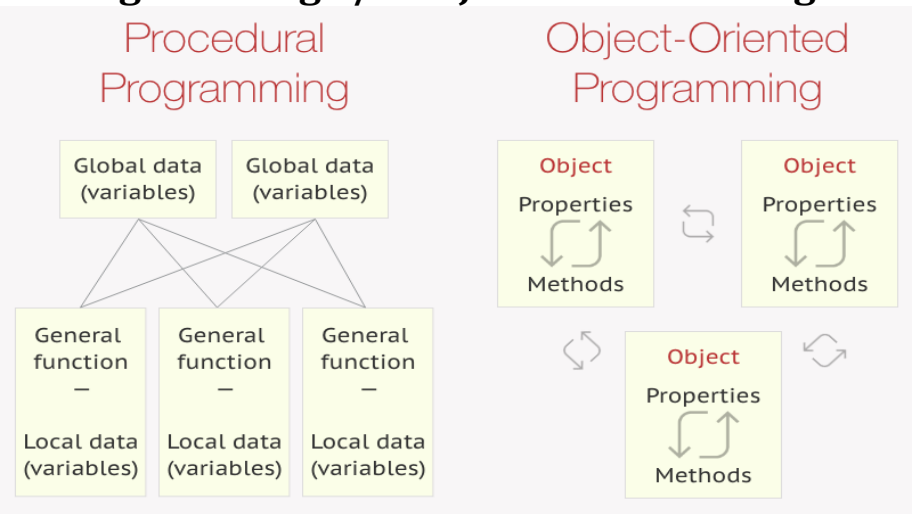
Person	University	Book	Student
• Name • Gender • Age • Height • Weight • Complexion • Color of Hair • Color of Skin • Profession	• Name • Establishment Year • Type • Principal • Courses Offered • Number of Students • Number of Faculties • Land Area • Built Up Area	• Title • Author • Publisher • Pages • ISBN • Type • Price • Edition • Volume	• Name • BirthDate • BloodGroup • Course • Semester • Mobile • Address • CGPA • ParentName
Person	**University**	**Book**	**Student**
• Eat() • Walk() • Run() • Talk() • Work() • ...	• DisplayUniName() • EnrollStudents() • DisplayPrincipalName() • ...	• EditBookStatus() • DisplayBookStatus() • DisplayPublisherName() • DisplayAuthorName() • ...	• RequestAdmission() • PayFees() • ExamRegistration() • ViewResult() • ...

Few of Real World Objects

Person: object	**University**: object	**Book**: object	**Student**: object
• Narendra Modi • Amitabh Bachhan • Sachin Tendulkar • Arijit Singh • Brad Pitt • R.G. Dhamsania	• Darshan University • Nirma University • Gujarat Technological University • Saurashtra University • Veer Narmad University	• Programming in C • The Secret • Two States • Bhagwad Gita • Ramayan • The Holy Bible	• Jack • Twinkle • John • Sita • Karan

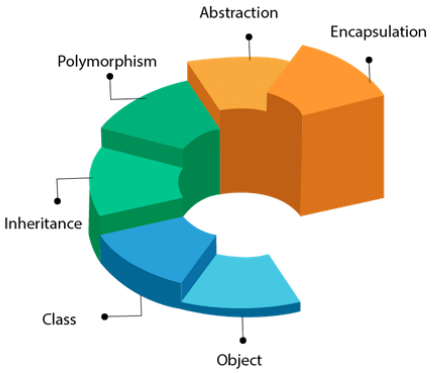
Cold Drinks: object	**Fan**: object	**Social Media**: object	**River**: object
• Coca Cola • Pepsi • Fanta • Spice • Sosyo	• Orient PSPO • Havells Galaxy • Bajaj Crest Neo • Usha Ex9 • Crompton	• Twitter • Whatsapp • Instagram • Facebook • Orkut	• Ganga • Narmada • Aji • Nile • Thames

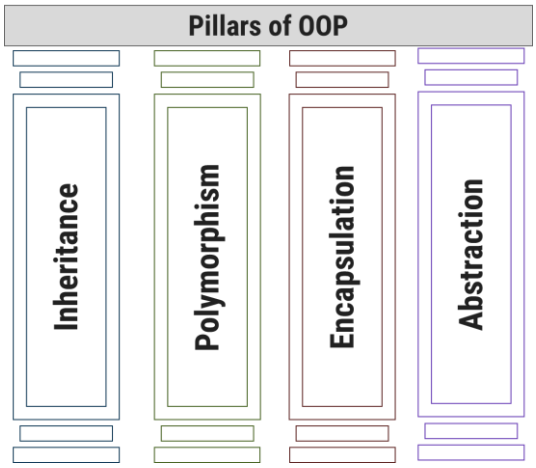
Procedural Programming v/s Object Oriented Programming



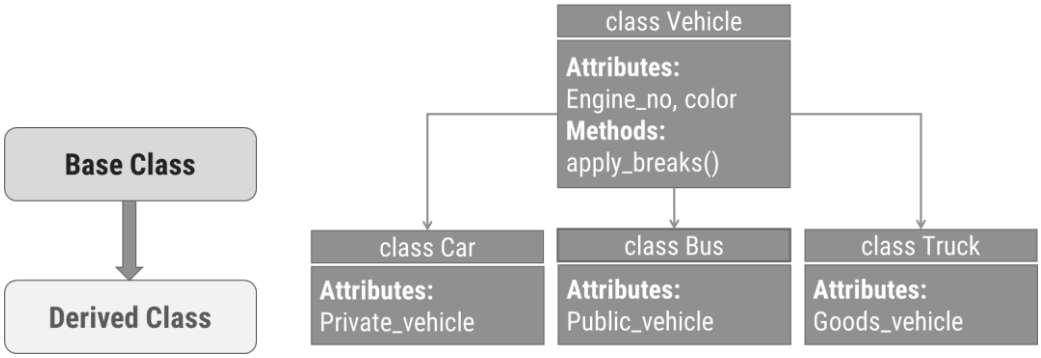
Procedural Programming	Object Oriented Programming
Program is divided into functions	Program is divided into classes & objects
The emphasis is on doing things	The emphasis is on data
Poor modeling to real world problems	Strong modeling to real world problems
Difficult to maintain large projects	Easy to maintain large projects
Poor data security	Strong data security
Code can't be reused in another project	Code can be reused across the projects
Not extensible	Extensible
Productivity is low	Productivity is high
Don't provide support for new data types	Provides support to new data types
Don't provide automatic memory management	Provides automatic memory management
e.g. Pascal, C, Basic, Fortran	e.g. C++, C#, Java

Principles of OOP





Inheritance

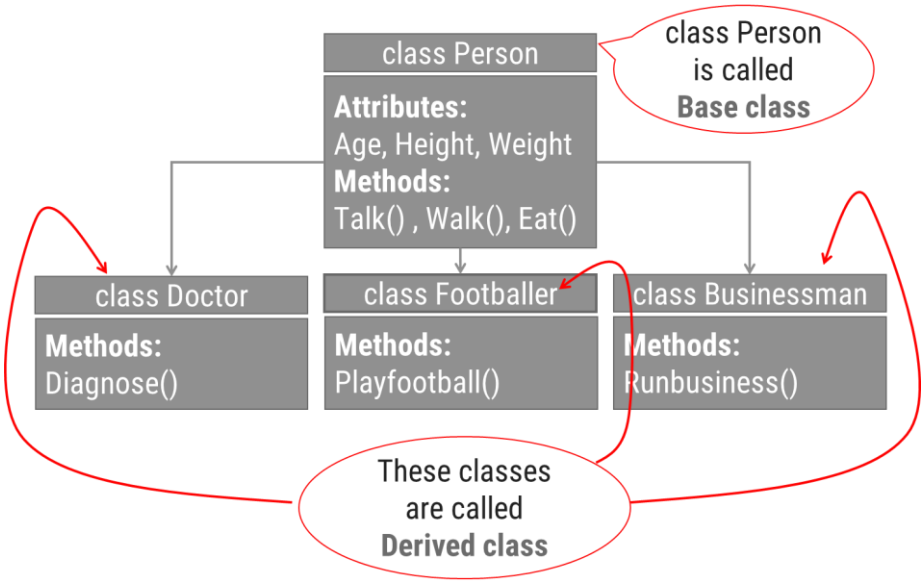


- The mechanism of a class to derive properties and characteristics from another class is called Inheritance.
- **Inheritance** is the process, by which a class can acquire(reuse) the properties and methods of another class.
- The mechanism of deriving a new class from an old class is called **inheritance**.
- The new class is called **derived class** and old class is called **base class**.
- It is the most important feature of Object Oriented Programming.
- **Base Class**: The class whose properties are inherited by sub class is called Base Class/Super class/Parent class.
- **Derived Class:** The class that inherits properties from another class is called Sub class/Derived Class/Child class.
- Inheritance is implemented using super class and sub class relationship in object-oriented languages.
- The derived class may have all the features of the base class and the programmer can add new features to the derived class.
- Inheritance is also known as "IS-A relationship" between parent and child classes.
- For Example :
    - Car **IS A** Vehicle
    - Bike **IS A** Vehicle
    - EngineeringCollege **IS A** College

- MedicalCollege **IS A** College
- MCACollege **IS A** College

Inheritance: Advantages

- Promotes reusability
- When an existing code is reused, it leads to less development and maintenance costs.
- It is used to generate more dominant objects.
- Avoids duplicity and data redundancy.
- Inheritance makes the sub classes follow a standard interface.



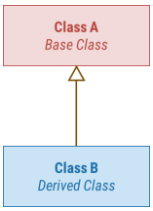
Implementing Inheritance

To inherit a class, you simply incorporate the definition of one class into another by using "**extends**" keyword.

Syntax:

```
class subclass-name extends superclass-name {
      // body of class…
}
```

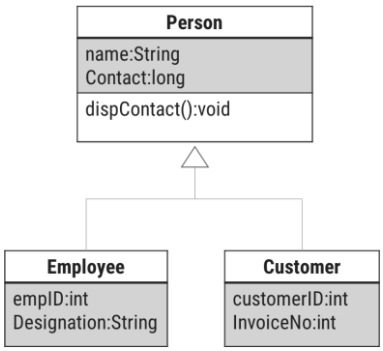
Implementing Inheritance in java




```
class A
{
  //SuperClass or ParentClass or BaseClass
}
```

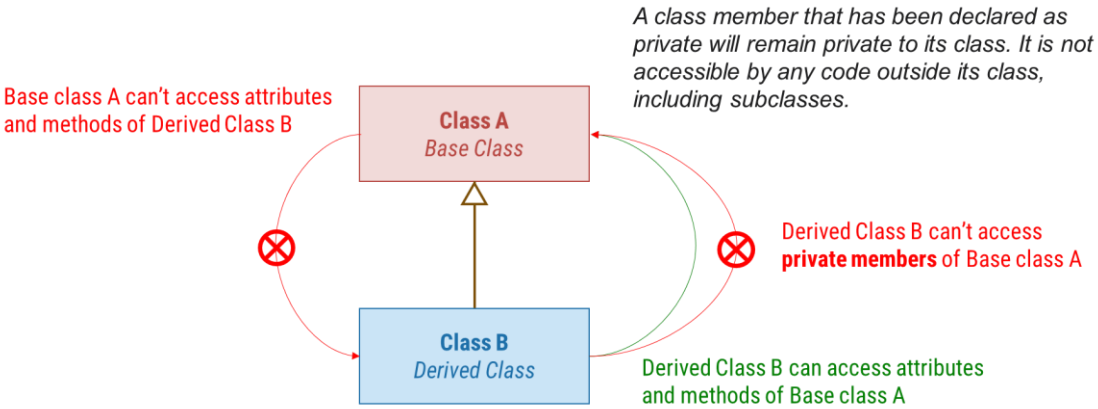
the keyword "extends" is used to create a subclass of A

```
class B extends A
{
  //SubClass or ChildClass or DerivedClass
}
```




```
1. class Person
2. {
3.         String name;
4.         long contact;
5.         public void  dispContact()   {
           System.out.println("num="+contact);
6.       }
7. }
8. class Employee extends Person
9. {
10.        int empID;
11.        String designation;
12.  }
13.  Class Customer extends Person
14.  {
15.        int customerID;
16.        int invoiceNo;
17.  }
```

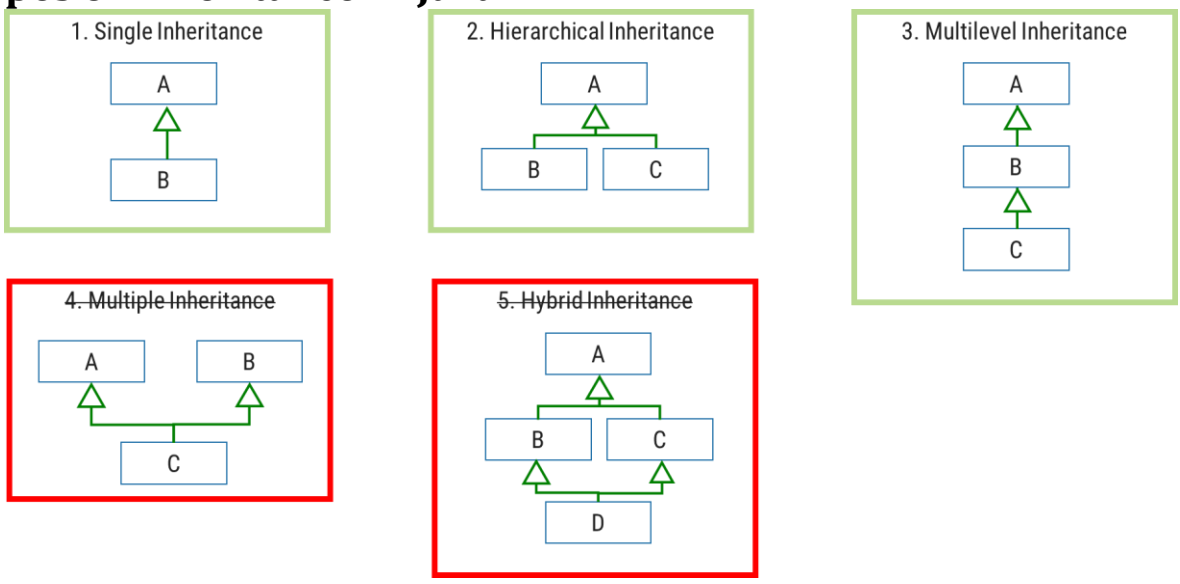
Property of Inheritance



InheritanceDemo.java

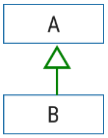
```
1. class A{
2.       protected int i;
3.       int j;
4.       void showij(){
5.       System.out.println("i="+i+" j="+j);
6.       }
7. }

8. class B extends A{ //inheritance
9.       int k;
10.            void showk(){
11.                  System.out.println("k="+k);
12.            }
13.            void add(){
         System.out.println("i+j+k="+(i+j+k));
14.            }
15.      }

16.  class InheritanceDemo{
17.      public static void main(String[] args)
18.        {
19.                  A superObjA= new A();
20.                  superObjA.i=10;
21.                  superObjA.j=20;
22.                  B subObjB= new B();
23.                  subObjB.k=30;
24.                  superObjA.showij();
25.                  subObjB.showk();
26.                  subObjB.add();
27.            }
28.      }
```

Types of Inheritance in Java



> ***Note:*** ***Multiple and Hybrid Inheritance is not supported in Java with the Class Inheritance, we can still use those Inheritance with Interface.***

Single Inheritance



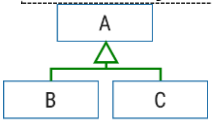
Single Inheritance Example

```
1. class A{
2.       protected int i;
3.       int j;
4.       void showij(){
5.       System.out.println("i="+i+" j="+j);
6.       }
7. }
8. class B extends A{ //inheritance
9.       int k;
10.            void showk(){
11.                  System.out.println("k="+k);
12.            }
13.            void add(){
         System.out.println("i+j+k="+(i+j+k));
14.            }
15.      }
16.  class InheritanceDemo{
17.      public static void main(String[]
                     args)
18.        {
19.                  A superObjA= new A();
20.                  superObjA.i=10;
21.                  superObjA.j=20;
22.                  B subObjB= new B();
23.                  subObjB.i=100
24.                  subObjB.j=100;
25.                  subObjB.k=30;
26.                  superObjA.showij();
27.                  subObjB.showk();
28.                  subObjB.add();
29.            }
30.      }
```

Hierarchical Inheritance

Hierarchical Inheritance Example

```
1. class A{
2.       protected int i;
3.       int j;
4.       void showij(){
5.            System.out.println("inside class A:i="+i+" j="+j);
6.       }
7. }
8. class B extends A{
9.       int k;
10.      void showk(){
11.           System.out.println("inside class B:k="+k);
12.      }
13.      void add_ijk(){
         System.out.println(i+"+"+j+"+"+k+"="+(i+j+k));
14.      }}
15.class C extends A{
16.      int m;
17.      void showm(){
18.           System.out.println("inside class C:k="+m);
19.      }
20.      void add_ijm(){
21.           System.out.println(i+"+"+j+"+"+m+"="+(i+j+m));
22.      }
23. }
24. class InheritanceLevel{
25.      public static void main(String[] args) {
26.           A superObjA= new A();
27.           superObjA.i=10;
28.           superObjA.j=20;
29.           superObjA.showij();
30.           B subObjB= new B();
31.           subObjB.i=100;
32.           subObjB.j=200;
33.           subObjB.k=300;
34.           subObjB.showk();
35.           subObjB.add_ijk();
36.           C subObjC= new C();
37.           subObjC.i=1000;
38.           subObjC.j=2000;;
39.           subObjC.m=3000;
40.           subObjC.showm();
41.           subObjC.add_ijm();
42.      }
43. }
```



Multilevel Inheritance

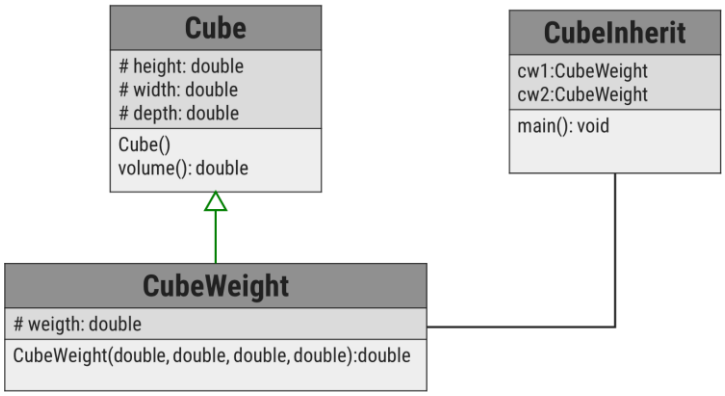
Multilevel Inheritance

```
1. class A{
2.       protected int i;
3.       int j;
4.       void showij(){
5.       System.out.println("inside class A:i="+i+" j="+j);
6.       }
7. }
8. class B extends A{
9.       int k;
10.            void showk(){
11.            System.out.println("inside class B:k="+k);
12.            }
13.            void add_ijk(){
         System.out.println(i+"+"+j+"+"+
         k+"="+(i+j+k));
14.            }}
15.  class C extends B{
16.            int m;
17.            void showm(){
18.            System.out.println("inside class C:k="+m);
19.            }
20.            void add_ijkm(){
21.            System.out.println(i+"+"+j+
         "+"+k+"+"+m+"="+(i+j+k+m));
22.            }
23.      }
24.  class InheritanceMultilevel{
25.            public static void main(String[] args) {
26.                  A superObjA= new A();
27.                  superObjA.i=10;
28.                  superObjA.j=20;
29.                  superObjA.showij();
30.                  B subObjB= new B();
31.                  subObjB.i=100;
32.                  subObjB.j=200;
33.                  subObjB.k=300;
34.                  subObjB.showk();
35.                  subObjB.add_ijk();
36.                  C subObjC= new C();
37.                  subObjC.i=1000;
38.                  subObjC.j=2000;
39.                  subObjC.k=3000;
40.                  subObjC.m=4000;
41.                  subObjC.showm();
42.                  subObjC.add_ijkm();
43.            }
44.      }
```

Output:

```
inside class A:i=10 j=20
inside class B:k=300
100+200+300=600
inside class C:k=4000
1000+2000+3000+4000=10000
```

Derived Class with Constructor



Derived Class with Constructor

```
1. class Cube{
2.       protected double height,width,depth;
3.       Cube(){
4.             System.out.println("inside default Constructor:CUBE");
5.       }
6.       double volume(){
7.             return height*width*depth;
8.       }
9. }
10. class CubeWeight extends Cube{
11.            double weigth;
12.            CubeWeight(double h,double w,double d, double m)
13.            {
14.            System.out.println("inside Constructor:CUBEWEIGTH");
15.                  height=h;C
16.                  width=w;
17.                  depth=d;
18.                  weigth=m;
19.            }}
20.  class CubeInherit{
21.            public static void main(String[] args) {
22.      CubeWeight cw1= new CubeWeight(10,10,10,20.5);
23.      CubeWeight cw2= new CubeWeight(100,100,100,200.5);
24.      System.out.println("cw1.volume()="  +cw1.volume());
25.      System.out.println("cw2.volume()="+cw2.volume());
26.        }}
```

Output:

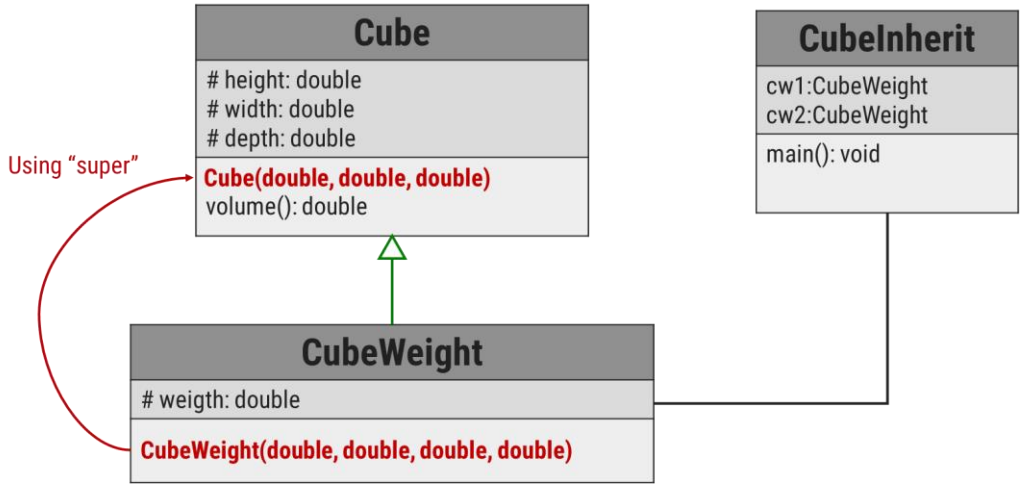
```
inside default Constructor:CUBE
inside Constructor:CUBEWEIGTH
inside default Constructor:CUBE
```

```
inside Constructor:CUBEWEIGTH
cw1.volume()=1000.0
cw2.volume()=1000000.0
```

Super Keyword

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.
- Super has two general forms:
  1. Calls the superclass constructor.
  2. Used to access a members (i.e. instance variable or method) of the superclass.

Using super to Call Superclass Constructors



Note: Call to super must be first statement in constructor

super to Call Superclass Constructors

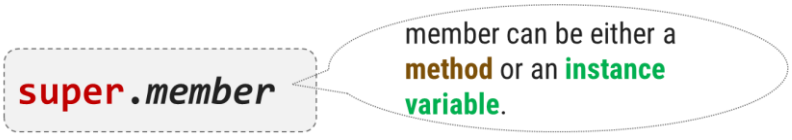
```
1. class Cube{
2.       protected double height,width,depth;
3.       Cube(double h,double w,double d){
4.             System.out.println("Constructor: CUBE");
5.             height=h;
6.             width=w;
7.             depth=d;
8.       }
9.       double volume(){
10.            return height*width*depth;
11.            }
12.      }
13. class CubeWeight extends Cube{
14.            double weigth;
15.            CubeWeight(double h,double w,double d, double m){
16.              super(h,w,d); //call superclassConstructor
17.              System.out.println("Constructor:CUBEWEIGTH");
18.              weigth=m;
19.            }
20.      }
21. class CubeInheritSuper{
22.        public static void main(String[] args) {
23.            CubeWeight cw1= new CubeWeight(10,10,10,20.5);
24.            CubeWeight cw2= new CubeWeight(100,100,100,200.5);
25.            System.out.println("cw1.volume()="+cw1.volume());
26.            System.out.println("cw1.weigth="+cw1.weigth);
27.            System.out.println("cw2.volume()="+cw2.volume());
28.            System.out.println("cw2.weigth="+cw2.weigth);
29.        }
30.      }
```

Output:

```
Constructor:CUBE
Constructor:CUBEWEIGTH
Constructor:CUBE
Constructor:CUBEWEIGTH
cw1.volume()=1000.0
cw1.weigth=20.5
cw2.volume()=1000000.0
cw2.weigth=200.5
```

Using super to access members

- The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used.



- This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Using super to access members

```
1. class A{
2.       int i;
3. }
4. class B extends A{
5.       int i,k;
6.       B(int a,int b){
7.            super.i=a;
8.            this.i=b;
9.       }
10.           void show(){
11.           System.out.println("super.i="+super.i);
12.           System.out.println("this.i="+this.i);
13.           }
14.      }
15.  class SuperMemberDemo{
16.      public static void main(String[] args)
17.        {
18.                  B b= new B(12,56);
19.                  b.show();
20.           }
21.      }
```

Output:
super.i=12
this.i=56

Points to remember for super

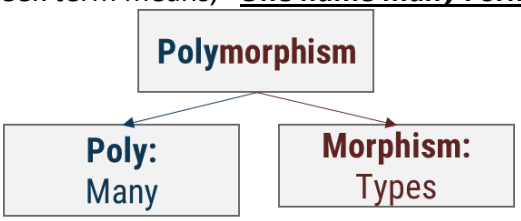
- When a subclass calls super( ), it is calling the constructor of its immediate superclass.
- This is true even in a multileveled hierarchy.
- super( ) must always be the first statement executed inside a subclass constructor.
- If a constructor does not explicitly call a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass.
- The most common application of super keyword is to eliminate the ambiguity between members of superclass and sub class.

Access Control

Access Modifier	Description
Private(-)	The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
Default(~)	The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
Protected(#)	The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
Public(+)	The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Polymorphism

- **Polymorphism**: It is a Greek term means, ***"One name many Forms".***

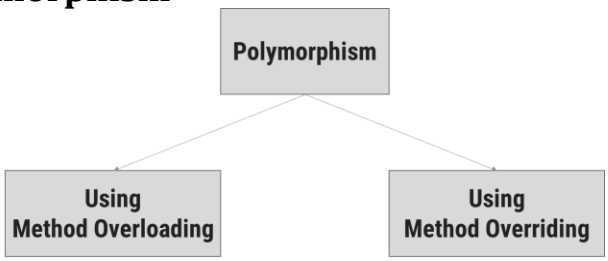


- Most important concept of object oriented programming
- In OOP, Polymorphism is the ability of an object to take on many forms.
- Polymorphism is the method in an object-oriented programming language that does different things depending on the class of the object which calls it.
- Polymorphism can be implemented using the concept of overloading and overriding.

Polymorphism: Advantages

- Single variable can be used to store multiple data types.
- Easy to debug the codes.
- It allows to perform a single act in different ways.
- Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding).
- Reduces coupling, increases reusability and makes code easier to read.

Implementing Polymorphism



Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.
- **Definition**: If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

Method Overriding Demo

```
1. class Shape{
2.       void draw(){
3.        System.out.println("Draw Shape");
4.       }
5. }
6. class Circle extends Shape{
7.       void draw(){
8.        System.out.println("Draw Circle");
9.       }
10.      }
11.class Square extends Shape{
12.            void draw(){
13.             System.out.println("Draw Square");
14.            }
15.      }
16.  class OverrideDemo{
17.            public static void main(String[] args) {
18.            Circle c= new Circle();
19.            c.draw(); //child class meth()
20.            Square sq= new Square();
21.            sq.draw();//child class meth()
22.            Shape sh= new Shape();
23.            sh.draw();//parentClass meth()
24.            }
25.      }
```

Output:

```
Draw Circle
Draw Square
Draw Shape
```

Why Overriding?

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism.
- By combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.
- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness.

Method Overriding: Points to remember

- Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.
- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Overloading vs Overriding

Method Overloading	Method Overriding
Overloading: Method with same name different signature	**Overriding:**Method with same name same signature
Known as Compile-time Polymorphism	Known as Run-time Polymorphism
It is performed *within class*.	It occurs *in two classes* with IS-A (inheritance) relationship.
Inheritance and method hiding is not involved here.	Here subclass method hides the super class method.

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

```
A a = new A(); //object of parent class
B b = new B(); //object of child class

A a = new B();//Up casting(Dynamic Method Dispatch)

B b= new A(); //Error! Not Allowed
```

Dynamic Method Dispatch Demo

```
1. class A{
2.  void display(){
3.       System.out.println("inside class A");
4.  }
5. }
6. class B extends A{
7. void display(){
8.       System.out.println("inside class B");
9.  }
10.      }
11.  class C extends A{
12.      void display(){
13.           System.out.println("inside class C");
14.       }
15.      }
16.class DispatchDemo{
17.           public static void main(String[] args) {
18.                A a = new A();
19.                B b = new B();
20.                C c = new C();
21.                A r; //obtain a reference of type A
22.                r=a;
23.                r.display();
24.                r=b;
25.                r.display();
26.                r=c;
27.                r.display();
28.           }
29.      }
```

Output:

```
inside class A
inside class B
inside class
```

"final" keyword

- The final keyword is used for restriction.
- final keyword can be used in many context
- Final can be:
  1. **Variable**
     If you make any variable as final, you cannot change the value of final variable (It will be constant).
  2. **Method**
     If you make any method as final, you cannot override it.
  3. **Class**
     If you make any class as final, you cannot extend it.

"final" as a variable

Cannot change the value of final variable.

```
public class FinalDemo {
 final int speedlimit=90;//final variable
 void run(){
      speedlimit=400; //ERROR!
 }
 public static void main(String args[]){
           FinalDemo obj=new  FinalDemo();
           obj.run();
 }
}
```

"final" as a method

If you make any method as final, you cannot override it.

```
class BikeClass{
  final void run(){
      System.out.println("Running Bike");
  }
}

class Pulsar extends BikeClass{
   void run(){
      System.out.println("Running Pulsar");//ERROR!
   }

   public static void main(String args[]){
      Pulsar p= new Pulsar();
      p.run();
   }
}
```

"final" as a Class

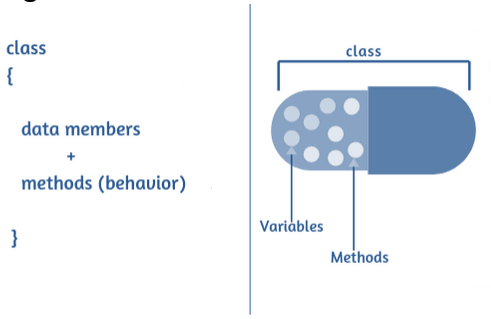
If you make any class as final, you cannot extend it.

```
final class BikeClass{
   void run(){
      System.out.println("Running Bike");
   }
}
class Pulsar extends BikeClass //ERROR!
{
   void run(){
      System.out.println("Running Pulsar");
   }

   public static void main(String args[]){
      Pulsar p= new Pulsar();
      p.run();
   }
}
```

Encapsulation

- The action of enclosing something in.
- In OOP, encapsulation refers to the bundling of data with the methods.

Data Members(e.g. int a=10)
+
Methods(e.g. add())



- The wrapping up of data and functions into a single unit is known as encapsulation
- The insulation of the data from direct access by the program is called data hiding or information hiding.
- It is the process of enclosing one or more details from outside world through access right.

Advantages

- Protects an object from unwanted access
- It reduces implementation errors.
- Simplifies the maintenance of the application and makes the application easy to understand.
- Protection of data from accidental corruption.

Abstraction

- Data abstraction is also termed as information hiding.
- Abstraction is the concept of object-oriented programming that "represents" only essential attributes and "hides" unnecessary information.
- Abstraction is all about representing the simplified view and avoid complexity of the system.
- It only shows the data which is relevant to the user.
- In object-oriented programming, it can be implemented using Abstract Class.

Advantages:

- It reduces programming complexity.
- Example:
  A car is viewed as a car rather than its numerous individual components.

Abstraction vs. Encapsulation

Abstraction	Encapsulation
It means act of removing/ withdrawing something unnecessary.	It is act of binding code and data together and keep the data secure from outside interference.
Applied at Designing stage.	Applied at Implementation stage.
E.g. Interface and Abstract Class	E.g. Access Modifier (public, protected, private)
Purpose: Reduce code complexity	Purpose: Data protection

Implementing Abstraction

- Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user.
- In other words, the user will have the information on what the object does instead of how the object will do it.
- Abstraction is achieved using Abstract classes and interfaces.
- A class which contains the abstract keyword in its declaration is known as abstract class.
    - Abstract classes may or may not contain abstract methods, i.e., methods without body (public void get();)
    - But, if a class has at least one abstract method, then the class must be declared abstract.
    - If a class is declared abstract, it cannot be instantiated.
    - To use an abstract class, we have to inherit it to another class and provide implementations of the abstract methods in it.

Abstract class

Abstract class (Example)

```
1. abstract class Car {
2.       public abstract double getAverage();
3. }
4. class Swift extends Car{
5.       public double getAverage(){
6.             return 22.5;
7.       }
8. }
9. class Baleno extends Car{
10.            public double getAverage(){
11.                  return 23.2;
12.            }
13.      }
14.      public class MyAbstractDemo{
15.            public static void main(String ar[]){
16.                  Swift s = new Swift();
17.                  Baleno b = new Baleno();
18.                  System.out.println(s.getAverage());
19.                  System.out.println(b.getAverage());
20.            }
21.      }
```

Why Abstract Class?

- Sometimes, we need to define a superclass that declares the structure of a given abstraction without providing a complete implementation.
- The superclass will only define a generalized form that will be shared by all the subclasses.
- The subclasses will fill the details of every method.
- When a superclass is unable to create a meaningful implementation for a method.

Points to remember for Abstract Class

- To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined.
- Cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

Interface

- An interface is similar to an abstract class with the following exceptions
  - All methods defined in an interface are abstract.
  - Interfaces doesn't contain any logical implementation
  - Interfaces cannot contain instance variables. However, they can contain public static final variables (ie. constant class variables)
- Interfaces are declared using the "interface" keyword
- Interfaces are more abstract than abstract classes
- Interfaces are implemented by classes using the "implements" keyword
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.

Interface: Syntax

public or not used(default)

Methods are without body(no implementation) and all methods are implicitly abstract.

implicitly final and static, cannot be changed by the implementing class, must be initialized with a constant value.

```
access interface name
{
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    type final-varname1 = value;
    type final-varname2 = value;
    // ...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}
```

```
access class classname
                [extends superclass]
                    [implements interface [,interface...]] {
// class-body
}
```

```
access interface name
{
        return-type method-name1(parameter-list);
        type final-varname1 = value;
 }
```

Interface (Example)

```
1. interface VehicleInterface {
2.       int a = 10;
3.       public void turnLeft();
4.       public void turnRight();
5.       public void accelerate();
6.       public void slowDown();
7. }
8. public class DemoInterface{
9.       public static void main(String[] a)
10.      {
11.            CarClass c = new CarClass();
12.            c.turnLeft();
13.      }
14. }
15. class CarClass implements VehicleInterface
16.      {
17.      public void turnLeft() {
18.      System.out.println("Left");
19.      }
20.      public void turnRight() {
21.      System.out.println("Right");
22.      }
23.      public void accelerate() {
24.      System.out.println("Speed");
25.      }
26.      public void slowDown() {
27.      System.out.println("Brake");
28.      }
29. }
```

Output:

```
Left
```

Interface: Partial Implementations

- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**.

Interface: Partial Implementations

```
1. interface VehicleInterface {
2. int a = 10;
3.              public void turnLeft();
4.              public void turnRight();
5.              public void accelerate();
6.              public void slowDown();
7. }
8. public class DemoInterface{
9.       public static void main(String[] a)
10.      {
11.             CarClass c = new CarClass();
12.             c.turnLeft();
13.      }
14.  }
15. abstract class CarClass implements VehicleInterface
16.  {
17.      public void turnLeft() {
18.            System.out.println("Left");
19.      }
20.      public void turnRight() {
21.            System.out.println("Right");
22.      }
23.  }
```

Interface:Example StackDemo.java

```
1. interface StackIntf{
2.         public void push(int p);
3.         public int pop();
4. }
5. class CreateStack implements StackIntf{
6.         int mystack[];
7.         int tos;
8.         CreateStack(int size){
9.               mystack= new int[size];
10.              tos=-1;
11.        }
12.        public void push(int p){
13.              if(tos==mystack.length-1){
14.                    System.out.println("StackOverflow");
15.              }
16.              else{
17.                    mystack[++tos]=p;
18.              }
19.        }
20.        public int pop(){
21.              if(tos<0){
22.                    System.out.println("StackUnderflow");
23.                    return 0;
24.        }
25.        else  return mystack[tos--];
26.        }
27. }
28. class StackDemo{
29.  public static void main(String[] args) {
30.        CreateStack cs1= new CreateStack(5);
31.        CreateStack cs2= new CreateStack(8);
32.        for(int i=0;i<5;i++)
33.              cs1.push(i);
34.        for(int i=0;i<8;i++)
              cs2.push(i);
36.        System.out.println("MyStack1=");
37.        for(int i=0;i<5;i++)
38.              System.out.println(cs1.pop());

39.        System.out.println("MyStack2=");

40.        for(int i=0;i<8;i++)
41.              System.out.println(cs2.pop());
42.        }
43. }
```

Interfaces Can Be Extended

- One interface can inherit another by use of the keyword extends.
- The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

InterfaceHierarchy.java

```
1. interface A{
2.       void method1();
3.        void method2();
4. }
5. interface B extends A{
6.       void method3();
7. }
8. interface C extends A{
9.       void method4();
10.      }
11.class InterfaceHierarchy{
12.      public static void main(String[] args) {
13.            MyClass1 c1=new MyClass1();
14.            MyClass2 c2=new MyClass2();
15.            c1.method1();
16.            c1.method2();
17.            c1.method3();
18.            c2.method1();
19.            c2.method2();
20.            c2.method4();
21.            }
22.      }
23. class MyClass1 implements B{
24.            public void method1(){
25.             System.out.println("inside MyClass1:method1()");}
26.
27.            public void method2(){
28.              System.out.println("inside MyClass1:method2()");
29.            }
30.
31.            public void method3(){
32.              System.out.println("inside MyClass1:method3()");
33.            }
34.      }
35.class MyClass2 implements C{
36.            public void method1(){
37.            System.out.println("inside MyClass2:method1()");}
38.
39.            public void method2(){
40.            System.out.println("inside MyClass2:method2()");
41.            }
42.
43.            public void method4(){
44.            System.out.println("inside MyClass2:method4()");
45.            }
46.      }
```

Interface: Points to Remember

- Any number of classes can implement an interface.
- One class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation.

Abstract class vs. Interface

Abstract class	Interface
Abstract class doesn't support **multiple inheritance.**	Interface **supports multiple inheritance**.
Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

instanceof operator

Syntax:

```
(Object reference variable ) instanceof (class/interface type)
```

Example:

```
boolean result = name instanceof String;
```

Wrapper classes

- A Wrapper class is a class whose object wraps or contains a primitive datatypes.
- When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive datatypes.
- In other words, we can wrap a primitive value into a wrapper class object.
- Use of wrapper class :
    - They convert primitive datatypes into objects.
    - The classes in java.util package handles only objects and hence wrapper classes help in this case also.
    - Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
    - An object is needed to support synchronization in multithreading.

Primitive datatypes: Wrapper class

Primitive datatype	*Wrapper class*	*Example*
byte	Byte	Byte b = new Byte((byte) 10);
short	Short	Short s = new Short((short) 10);
int	Integer	Integer i = new Integer(10);
long	Long	Long l = new Long(10);
float	Float	Float f = new Float(10.0);
double	Double	Double d = new Double(10.2);
char	Character	Character c = new Character('a');
boolean	Boolean	Boolean b = new Boolean(true);

Parsing the String

Using wrapper class we can parse string to any primitive datatype (Except char).

```
byte b1 =       Byte.parseByte("10");
short s =       Short.parseShort("10");
int i =         Integer.parseInt("10");
long l =        Long.parseLong("10");
float f =       Float.parseFloat("10.5");
double d =      Double.parseDouble("10.5");
boolean b2 =    Boolean.parseBoolean("true");
char c =        Character.parseCharacter('a');
```

BigInteger and BigDecimal

- The BigInteger class found in java.math package is used for mathematical operation which involves very big integer calculations that are outside the limit of all available primitive data types.
    - For example factorial of 100 contains 158 digits in it so we can't store it in any primitive data type available.
    - There is no theoretical limit on the upper bound of the range because memory is allocated dynamically

```
import java.math.BigInteger;
public class DemoBigNumbers {
 public static void main(String[] args) {
  BigInteger bi = new BigInteger("12345678912345678912345678890");
  System.out.println(bi); // will return 12345678912345678912345678890
     }
}
```

```
BigDecimal bd = new BigDecimal("111111.000000000000000000025");
System.out.println(bd); //111111.000000000000000000025
```

- The BigDecimal class found in java.math package provides operation for arithmetic, comparison, hashing, rounding, manipulation and format conversion.
    - This method can handle very small and very big floating point numbers with great precision.
- An object of the String class represents a string of characters.

- The String class belongs to the java.lang package, which does not require an import statement.
- Like other classes, String has constructors and methods.
- String class has two operators, + and += (used for concatenation).

Empty String :

- An empty String has no characters. Its length is 0.

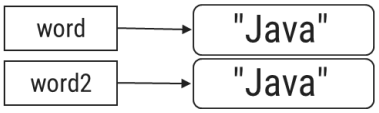
```
String word1 = "";
String word2 = new String();
```

- Not the same as an uninitialized String.

```
String word1;
```

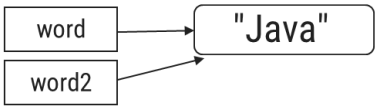
- Copy constructor creates a copy of an existing String.

```
String word = new String("Java");
String word2 = new String(word);
```



Assignment: Both variables point to the same String.

```
String word = "Java";
String word2 = word;
```



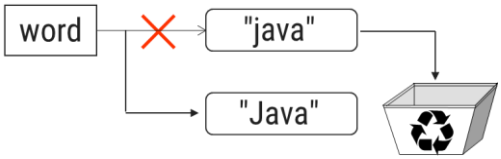
String Class

- In java, a string is a sequence of characters. But, unlike many other languages that implement strings as character arrays, java implements strings as objects of type String.
- When we create a String object, we are creating a string that cannot be changed. That is, once a String object has been created, we cannot change the characters that comprise that string. We can perform all types of operations.
- For those cases in which a modifiable string is desired, java provides two options: StringBuffer and StringBuilder. Both hold strings that can be modified after they are created.
- The String, StringBuffer and StringBuilder classes are defined in java.lang. Thus, they are available to all programs automatically. All three implements CharSequence interface.
- **String Constructor**
- The String class support several constructors. To create an empty String, you call the default constructor.
- For example,

  String s=new String();
  this will create an instance of with no characters in it.
  String s = new String("Computer Deparatment");

```
class StringEx
{
 public static void main(String args[])
 {
       String s1=new String("Computer Department");
       String s2;
       s2=s1 + "Darshan University";
       System.out.println(s2);
 }
}
```

String Immutability

Advantage	Disadvantage
Convenient — Immutable objects are convenient because several references can point to the same object safely.	Less efficient — you need to create a new string and throw away the old one even for small changes.
`String name="DIET - Rajkot";` *`foo(name);`* `//Some operations on String name` `name.substring(7,13);` *`bar(name);`*	`String word = "java";` `char ch = Character.toUpperCase(` `word.charAt (0));` `word = ch + word.substring (1);` 

String Methods

Method Call	Task Performed
S2=s1.toLowerCase;	Conver ts the string s1 to lowercase
S2=s1.toUpperCase;	Converts the string s1 to uppercase
S2=s1.replace('x','y')	Replace all appearances of x with y.
S2=s1.trim()	Remove white spaces at the beginning and end of the string s1
S1.equals(s2)	Returns true if s1 and s2 are equal
S1.equalsIgnoreCase(s2)	Returns true if s1=s2, ignoring the case of characters
S1.length()	Gives the length of s1
S1.CharAt(n)	Gives the nth character of s1
S1.compareTo(s2)	Returns –ve if s1<s2, +ve if s1>s2, and 0 if s1=s2
S1.concat(s2)	Concatenates s1 and s2
S1.substring(n)	Gives substring starting from nth character.
S1.substring(n,m)	Gives substring starting from nth char up to mth
String.ValueOf(p)	Returns the string representation of the specified type argument.

toString()	This object (which is already a string!) is itself returned.
S1.indexOf('x')	Gives the position of the first occurrence of 'x' in the string s1
S1.indexOf('x',n)	Gives the position of 'x' that occurs after nth position in the string s1
String.ValueOf(variable)	Converts the parameter value of string representation

StringBuffer

- The java.lang.StringBuffer class is a thread-safe, mutable sequence of characters.
- Following are the important points about StringBuffer:
  - A string buffer is like a String, but can be modified (mutable).
  - It contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.
  - They are safe for use by multiple threads.
- StringBuffer defines these Constructor:

```
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

- Remember : "StringBuffer" is mutable
- As StringBuffer class is mutable, we need not to replace the reference with a new reference as we have to do it with String class.

```
StringBuffer str1 = new StringBuffer("Hello Everyone");
str1.reverse();
// as it is mutable can not write str1 = str1.reverse();
// it will change to value of the string itself
System.out.println(str1);
// Output will be "enoyrevE olleH"
```

StringBuffer Methods

Method	Description
append(String s)	Used to append the specified string with this string.
insert(int offset, String s)	Used to insert the specified string with this string at the specified position.
replace(int startIndex, int endIndex, String str)	Used to replace the string from specified startIndex and endIndex.
delete(int startIndex, int endIndex)	Used to delete the string from specified startIndex and endIndex.
reverse()	Used to reverse the string.

String Builder

- Java StringBuilder class is used to create mutable string.
- The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized.
- It is available since JDK 1.5.
- It has similar methods as StringBuffer like append, insert, reverse etc…

ArrayList

- The java.util.ArrayList class provides resizable-array and implements the List interface.
- Following are the important points about ArrayList:
- It implements all optional list operations and it also permits all elements, including null.
- It provides methods to manipulate the size of the array that is used internally to store the list.

ArrayList (constructors) :

Sr	Constructor & Description
1	ArrayList() This constructor is used to create an empty list with an initial capacity sufficient to hold 10 elements.
2	ArrayList(Collection<? extends E> c) This constructor is used to create a list containing the elements of the specified collection.
3	ArrayList(int initialCapacity) This constructor is used to create an empty list with an initial capacity.

ArrayList (method)

Sr	*Method & Description*
1	void add(int index, E element) This method inserts the specified element at the specified position in this list.
2	boolean addAll(Collection<? extends E> c) This method appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator
3	void clear() This method removes all of the elements from this list.
4	boolean contains(Object o) This method returns true if this list contains the specified element.
5	E get(int index) This method returns the element at the specified position in this list.
6	int indexOf(Object o) This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
7	**boolean isEmpty()** **This method returns true if this list contains no elements.**
8	int **lastIndexOf**(Object o)

	This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
9	boolean **remove**(Object o) This method removes the first occurrence of the specified element from this list, if it is present.
10	E **set**(int index, E element) This method replaces the element at the specified position in this list with the specified element.
11	int **size**() This method returns the number of elements in this list.
12	Object[] **toArray**() This method returns an array containing all of the elements in this list in proper sequence (from first to last element).

Object Oriented Programming -I (3140705)

Dedicated Faculty, Committed Education

Darshan

Institute of Engineering & Technology

Unit-06

Exception Handling

Prof. Jayesh D. Vagadiya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot
Jayesh.vagadiya@darshan.ac.in
9537133260

Outline

- ✓ Exception
- ✓ Using try and catch
- ✓ Nested try statements
- ✓ The Exception Class Hierarchy
- ✓ throw statement
- ✓ throws statement

Exceptions

- An **exception** is an object that describes an **unusual** or **erroneous situation**.
- Exceptions are thrown by a program, and may be caught and handled by another part of the program.
- A program can be separated into a **normal execution flow** and an **exception execution flow**.
- An error is also represented as an object in Java, but usually represents a unrecoverable situation and should not be caught.
- Java has a predefined set of exceptions and errors that can occur during execution.
- A program can deal with an exception in one of three ways:
  - **ignore** it
  - **handle** it where it occurs
  - handle it at **another place** in the program
- The manner in which an exception is processed is an important design consideration.

Using try and catch

- Example:

```
try{
        // code that may cause exception
}
catch(Exception e){
        // code when exception occurred
}
```

Exception is the superclass of all the exception that may occur in Java

- Multiple catch:

```
try{
        // code that may cause exception
}
catch(ArithmeticException ae){
        // code when arithmetic exception occurred
}
catch(ArrayIndexOutOfBoundsException aiobe){
        // when array index out of bound exception occurred
}
```

Nested try statements

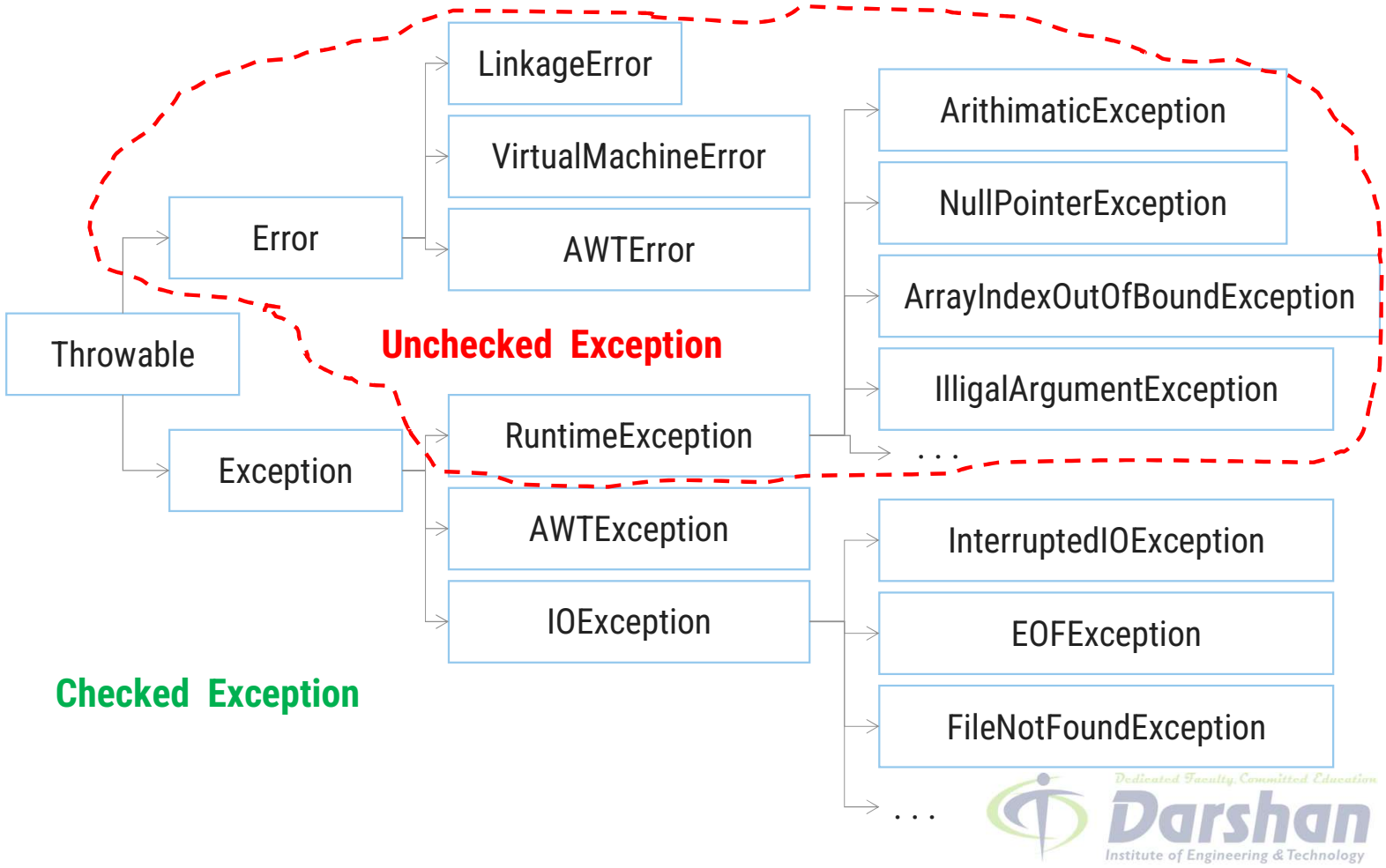
```
try
{
    try
    {
        // code that may cause array index out of bound exception
    }
    catch(ArrayIndexOutOfBoundsException aiobe)
    {
        // code when array index out of bound exception occured
    }
    // other code that may cause arithmetic exception
}
catch(ArithmeticException ae)
{
        // code when arithmetic exception occurred
}
```

Types of Exceptions

- An exception is either checked or unchecked.
- **Checked** Exceptions
  - A checked exception either must be caught by a method, or must be listed in the throws clause of any method that may throw or propagate it.
  - The compiler will issue an error if a checked exception is not caught or asserted in a throws clause
  - Example: IOException, SQLException etc…
- **Unchecked** Exceptions
  - An unchecked exception does not require explicit handling, though it could be processed using try catch.
  - The only unchecked exceptions in Java are objects of type RuntimeException or any of its descendants.
  - Example: ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException etc..

The Exception Class Hierarchy

- Classes that define exceptions are related by inheritance, forming an exception class hierarchy.
- All error and exception classes are descendents of the Throwable class
- The custom exception can be created by extending the Exception class or one of its descendants.



Java's Inbuilt Unchecked Exceptions

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.

Java's Inbuilt Checked Exceptions

Exception	Meaning
ClassNotFoundException	Class not found.
IOException	Input Output Exceptions
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

throw statement

- it is possible for your program to throw an exception **explicitly**, using the **throw** statement.
- The general form of throw is shown here:

  throw *ThrowableInstance;*
- Here, ***ThrowableInstance*** *must be an object of type* ***Throwable*** *or a* ***subclass*** *of Throwable.*
- Primitive types, such as int or char, as well as non-throwable classes, such as String and Object, cannot be used as exceptions.
- There are two ways you can obtain a Throwable object:
  - using a parameter in a catch clause,
  - or creating one with the new operator.

Throw (Example)

```
public class DemoException {
    public static void main(String[] args) {
        int balance = 5000;

        Scanner sc = new Scanner(System.in);
        System.out.println("Enter Amount to withdraw");
        int withdraw = sc.nextInt();
        try {
            if(balance - withdraw < 1000) {
                throw new Exception("Balance must be grater than 1000");
            }
            else {
                balance = balance - withdraw;
            }
        }catch(Exception e) {
                e.printStackTrace();
        }
    }
}
```

The finally statement

- The purpose of the **finally** statement will allow the execution of a segment of code regardless if the try statement throws an exception or executes successfully
- The advantage of the **finally** statement is the ability to clean up and release resources that are utilized in the **try** segment of code that might not be released in cases where an exception has occurred.

```
public class MainCall {
    public static void main(String args[]) {
        int balance = 5000;
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter Amount to withdraw");
        int withdraw = sc.nextInt();
        try {
            if(balance - withdraw < 1000) {
               throw new Exception("Balance < 1000 error");
            }
            else {
               balance = balance - withdraw;
            }
        }catch(Exception e) {
            e.printStackTrace();
        }
        finally {
            sc.close();
        }
    }
}
```

throws statement

- A throws statement lists the types of exceptions that a **method** might throw.
- This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.
- This is the general form of a method declaration that includes a **throws clause:**

  type method-name(parameter-list) ***throws*** *exception-list* {

  // body of method

  }
- Here, *exception-list is a comma-separated list of the exceptions that a method can throw.*
- Example :

```
void myMethod() throws ArithmeticException, NullPointerException
{
    // code that may cause exception
}
```

Create Your Own Exception

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable).
- The Exception class does not define any methods of its own. It does inherit those methods provided by Throwable.
- Thus, all exceptions have methods that you create and defined by Throwable.

Methods of Exception class

Method	Description
Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement.
Throwable initCause(Throwable causeExc)	Associates causeExc with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream stream)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement elements[ ])	Sets the stack trace to the elements passed in elements.
String toString( )	Returns a String object containing a description of the exception.

Custom Exception (Example)

```
// A Class that represents
user-defined exception
class MyException
        extends Exception
{
  public MyException(String s)
  {
      // Call constructor of
        parent (Exception)
        super(s);
  }
}
```

```
class MainCall {
    static int currentBal = 5000;
    public static void main(String args[]) {
        try {
                int amount = Integer.parseInt(args[0]);
                withdraw(amount);
        } catch (Exception ex) {
            System.out.println("Caught");
            System.out.println(ex.getMessage());
        }
    }
    public static void withdraw(int amount) throws Exception
    {
        int newBalance = currentBal - amount;
        if(newBalance<1000) {
                throw new MyException("Darshan Exception");
        }
    }
}
```

Object Oriented Programming -I (3140705)

Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

Unit-07

JavaFX and Event-driven programming and animations

Prof. Jayesh D. Vagadiya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot
Jayesh.vagadiya@darshan.ac.in
9537133260

Outline

- ✓ What is JavaFX?
- ✓ Architecture of JavaFX API
- ✓ JavaFX Application Structure
- ✓ Lifecycle of JavaFX Application
- ✓ 2D Shape
- ✓ JavaFX - Colors
- ✓ JavaFX – Image
- ✓ Layout Panes
- ✓ JavaFX – Events
- ✓ Property Binding
- ✓ Animation

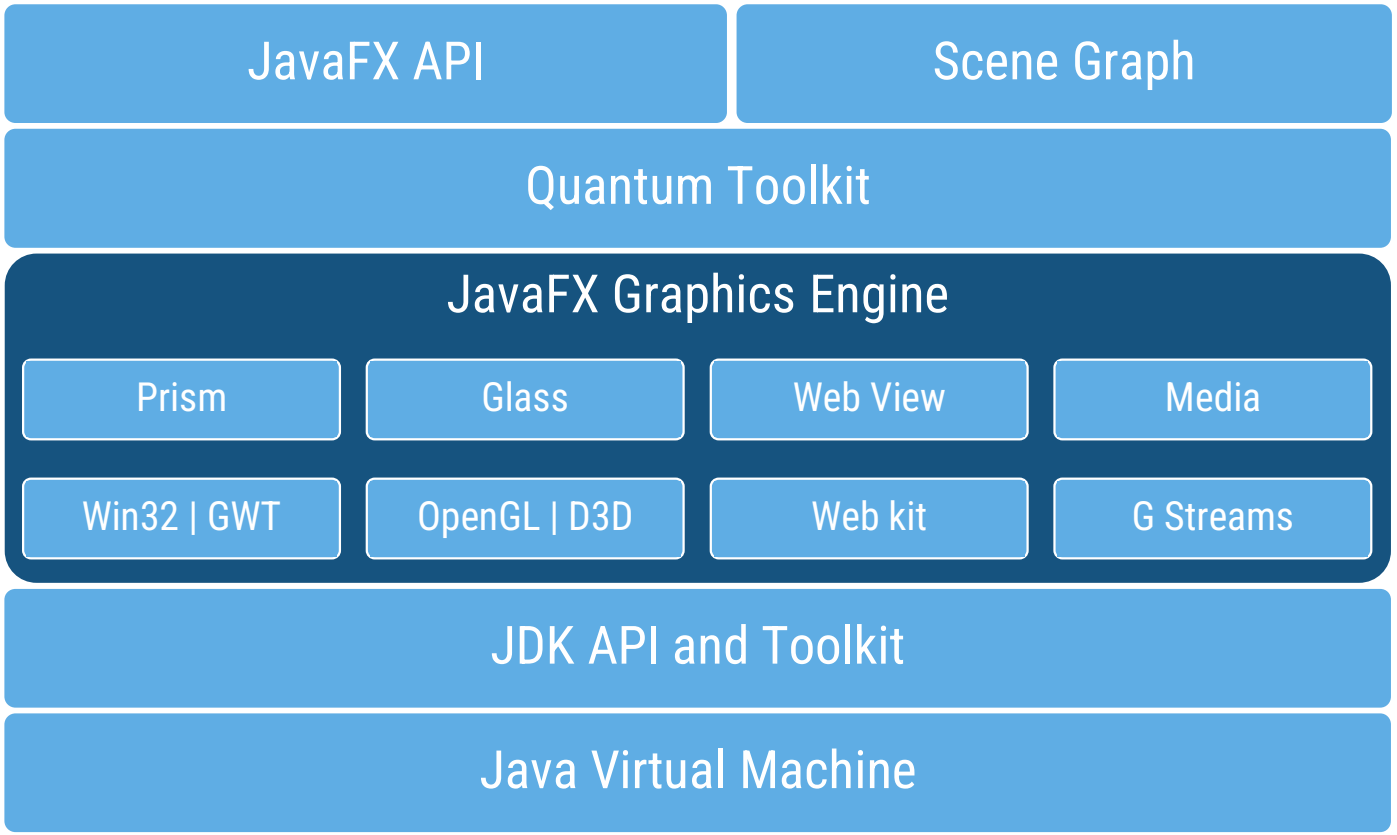
What is JavaFX?

- *JavaFX* is a Java library used to build Rich Internet Applications (RIA) and Desktop Applications.
- The applications developed using JavaFX can run on various devices such as Desktop Computers, Mobile Phones, TVs, Tablets, etc.
- To develop GUI Applications using Java programming language, the programmers rely on libraries such as Advanced Windowing Toolkit (AWT) and Swing. After the advent of JavaFX, these Java programmers can now develop GUI applications effectively with rich content.
- Why we need JavaFX
  - To develop Client Side Applications with rich features, the programmers used to depend on various libraries to add features such as Media, UI controls, Web, 2D and 3D, etc.
  - JavaFX provides a rich set of graphics and media API's and it leverages the modern Graphical Processing Unit through hardware accelerated graphics.
  - One can use JavaFX with JVM based technologies such as Java, Groovy and JRuby. If developers opt for JavaFX, there is no need to learn additional technologies.

Features of JavaFX

- Written in Java
- FXML
- Scene Builder
- Swing Interoperability
- Built-in UI controls
- CSS like Styling
- Canvas and Printing API
- Rich set of API's
- Integrated Graphics library
- Graphics pipeline

Architecture of JavaFX API



Architecture of JavaFX API (Cont.)

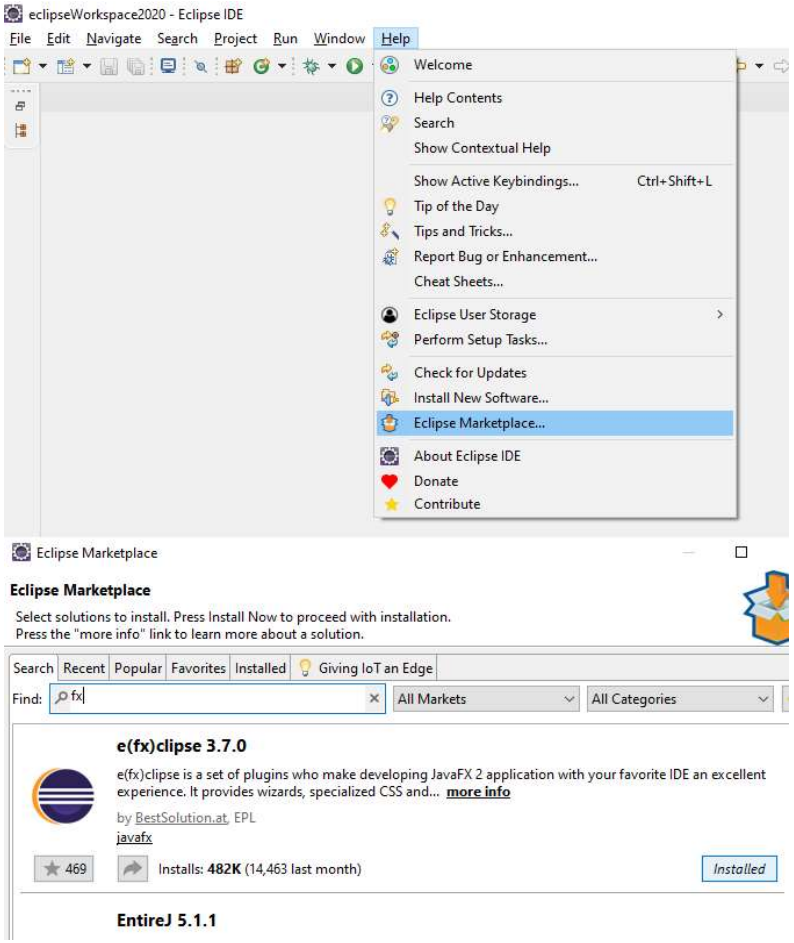
- Scene Graph
  - A Scene Graph is the starting point of the construction of the GUI Application. It holds the (GUI) application primitives that are termed as nodes.
  - A node is a visual/graphical object and it may include
    - Geometrical (Graphical) objects
    - UI controls
    - Containers
    - Media elements
- Prism
  - Prism is a high performance hardware–accelerated graphical pipeline that is used to render the graphics in JavaFX. It can render both 2-D and 3-D graphics.
- GWT (Glass Windowing Toolkit)
  - GWT provides services to manage Windows, Timers, Surfaces and Event Queues.
  - GWT connects the JavaFX Platform to the Native Operating System.

Architecture of JavaFX API (Cont.)

- Quantum Toolkit
  - It is an abstraction over the low-level components of Prism, Glass, Media Engine, and Web Engine. It ties Prism and GWT together and makes them available to JavaFX.
- WebView
  - WebView is the component of JavaFX which is used to process HTML content. It uses a technology called Web Kit, which is an internal open-source web browser engine. This component supports different web technologies like HTML5, CSS, JavaScript, DOM and SVG.
- Media Engine
  - The JavaFX media engine is based on an open-source engine known as a Streamer. This media engine supports the playback of video and audio content.

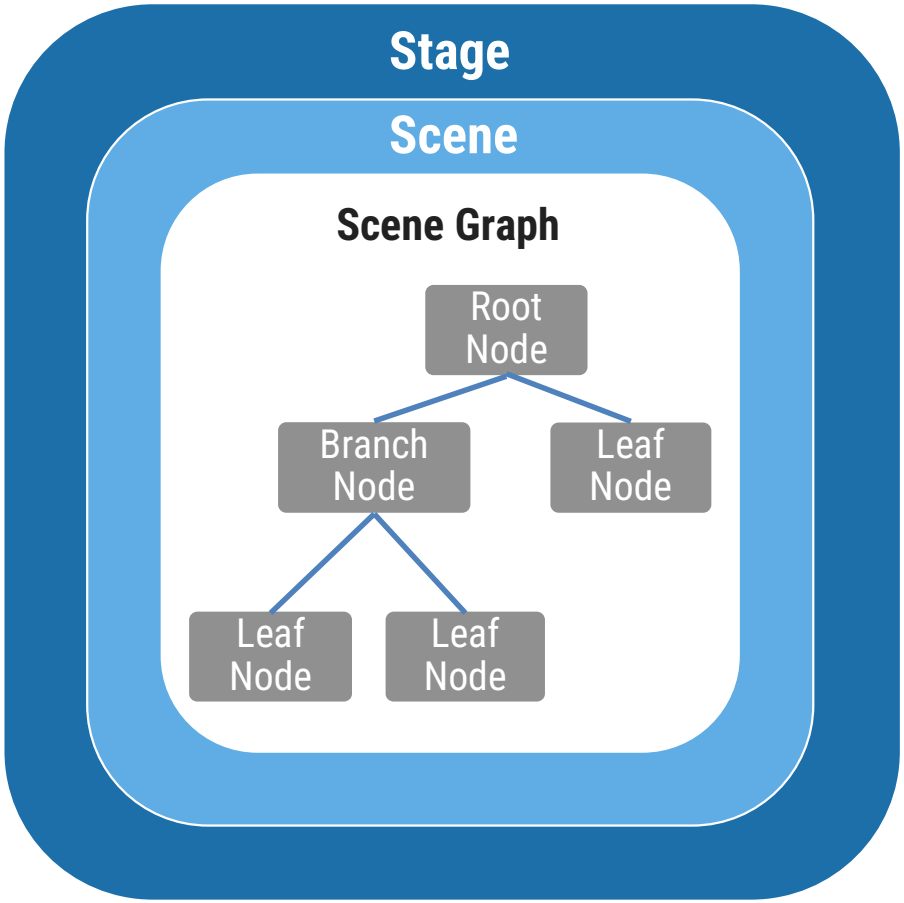
Installing JavaFX

- You can install JavaFX directly on eclipse using the Market Place.
- To download JavaFX from Market Place, open **`Help -> Eclipse Market Place`**
- Then simply search "**fx**" and install the e(fx)clipse from the Market Place.
- It will take time to install and after installation you need to restart the eclipse.
- Alternatively you can also download JavaFX SDK manually from https://gluonhq.com/products/javafx/
- Then you need to create a custom library in eclipse to load JavaFX to your project.



Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

JavaFX Application Structure



Stage

- A stage (a window) contains all the objects of a JavaFX application.
- It is represented by `Stage` class of the package `javafx.stage`.
- The primary stage is created by the platform itself. The created stage object is passed as an argument to the `start()` method of the `Application` class.
- A stage has two parameters determining its position namely Width and Height.
- There are five types of stages available
  - Decorated `// stageObj.initStyle(StageStyle.DECORATED);`
  - Undecorated `// stageObj.initStyle(StageStyle.UNDECORATED);`
  - Transparent `// stageObj.initStyle(StageStyle.TRANSPARENT);`
  - Unified `// stageObj.initStyle(StageStyle.UNIFIED);`
  - Utility `// stageObj.initStyle(StageStyle.UTILITY);`
- You have to call the `show()` method to display the contents of a stage.

Scene

- A scene represents the physical contents of a JavaFX application. It contains all the contents of a scene graph.
- The class `Scene` of the package `javafx.scene` represents the scene object. At an instance, the scene object is added to only one stage.
- You can create a scene by instantiating the `Scene` Class.
- You can opt for the size of the scene by passing its dimensions (height and width) along with the root node to its constructor.

Scene Graph and Nodes

- A scene graph is a tree-like data structure (hierarchical) representing the contents of a scene. In contrast, a node is a visual/graphical object of a scene graph. A node may include,
  - Geometrical (Graphical) objects (2D and 3D) such as – Circle, Rectangle, Polygon, etc.
  - UI Controls – Button, Checkbox, Choice Box, Text Area, etc.
  - Containers (Layout Panes) – Border Pane, Grid Pane, Flow Pane, etc.
  - Media elements – Audio, Video and Image Objects.
- The `Node` class of the package `javafx.scene` represents a node in JavaFX, this class is the super class of all the nodes.
  - Root Node – First Scene Graph is known as Root node. It's mandatory to pass root node to the scene graph.
  - Branch Node/Parent Node – The node with child nodes are known as branch/parent nodes. The abstract class named `Parent` of the package `javafx.scene` is the base class of all the parent nodes, and those parent nodes will be of the following types
    - Group – A group node is a collective node that contains a list of children nodes.
    - Region – It is the base class of all the JavaFX Node based UI Controls, such as Chart, Pane and Control.
    - WebView – This node manages the web engine and displays its contents.
  - Leaf Node – The node without child nodes is known as the leaf node. For example, Rectangle, Ellipse, Box, ImageView, MediaView are examples of leaf nodes.

Steps to create JavaFX application

1. Extend the Application class of `javafx.application` package in your class.
2. Override `start` method of Application class.
3. Prepare a scene graph with the required nodes.
4. Prepare a Scene with the required dimensions and add the scene graph (root node of the scene graph) to it.
5. Prepare a stage and add the scene to the stage and display the contents of the stage.

3. Prepare a Scene graph

- Since the root node is the first node, you need to create a root node and it can be chosen from the *Group, Region or WebView.*
  - Group

    A Group node is represented by the class named `Group` which belongs to the package `javafx.scene`, you can create a Group node by instantiating this class as shown below.

    ```
    Group root = new Group();
    Group root = new Group(NodeObject);
    ```

  - Region

    It is the Base class of all the JavaFX Node-based UI Controls, such as –

    - `Chart` – This class is the base class of all the charts and it belongs to the package `javafx.scene.chart` which embeds charts in application.
    - `Pane` – A Pane is the base class of all the layout panes such as AnchorPane, BorderPane, DialogPane, etc. This class belong to a package that is called as – `javafx.scene.layout` which inserts predefined layouts in your application.
    - `Control` – It is the base class of the User Interface controls such as Accordion, ButtonBar, ChoiceBox, ComboBoxBase, HTMLEditor, etc. This class belongs to the package javafx.scene.control.

  - WebView

    This node manages the web engine and displays its contents.

4. Preparing the Scene

- A JavaFX scene is represented by the `Scene` class of the package `javafx.scene`.

```
Scene scene = new Scene(root,width,heigth);
```

- While instantiating, it is mandatory to pass the root object to the constructor of the `Scene` class whereas width and height of the scene are optional parameters to the constructor.

5. Preparing the Stage

- Stage is the container of any JavaFX application and it provides a window for the application. It is represented by the `Stage` class of the package `javafx.stage`.
- An object of this class is passed as a parameter of the `start()` method of the Application class.
- Using this object, various operations on the stage can be performed like
  - *Set the title* for the stage using the method `setTitle()`.

```
primaryStage.setTitle("Sample application");
```

  - *Attach the scene* object to the stage using the `setScene()` method.

```
primaryStage.setScene(scene);
```

  - *Display the contents* of the scene using the `show()` method as shown below.

```
primaryStage.show();
```

Basic Example of JavaFX

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class MyFirstGUI extends Application{
    public void start(Stage primaryStage) throws Exception{
        Group root = new Group();
        Scene s = new Scene(root,600,400);
        primaryStage.setTitle("My First User Interface");
        s.setFill(Color.ORANGE);
        primaryStage.setScene(s);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Create Scene Graph using Group, Region or WebView

Create Scene by adding Group (root) to it along with its width and height

Set the scene to the stage object (primaryStage) which is passed as an argument to start() using setScene() method

Lifecycle of JavaFX Application

- The JavaFX `Application` class has three life cycle methods.

  `init()` – An empty method which can be overridden, stage or scene cannot be created in this method.

  `start()` – The entry point method where the JavaFX graphics code is to be written.

  `stop()` – An empty method which can be overridden, here the logic to stop the application is written.

- It also provides a static method named `launch()` to launch JavaFX application. This method is called from static content only mainly in main method.
- Whenever a JavaFX application is launched, the following actions will be carried out (in the same order).
  - An instance of the application class is created.
  - `init()` method is called.
  - `start()` method is called.
  - The launcher waits for the application to finish and calls the `stop()` method.

2D Shape

- 2D shape is a geometrical figure that can be drawn on the XY plane like Line, Rectangle, Circle, etc.
- Using the JavaFX library, you can draw –
  - Predefined shapes – Line, Rectangle, Circle, Ellipse, Polygon, Polyline, Cubic Curve, Quad Curve, Arc.
  - 2D shape by parsing SVG path.
- Each of the above mentioned 2D shape is represented by a class which belongs to the package `javafx.scene.shape`. The class named Shape is the base class of all the 2-Dimensional shapes in JavaFX.

Classes for Shape (`javafx.scene.shape`)

Shape	Class	Example
Line	`Line`	`Line line = new Line();` `line.setStartX(100.0);` `line.setStartY(150.0);` `line.setEndX(500.0);` `line.setEndY(150.0);`
Rectangle & Rounded Rectangle	`Rectangle`	`Rectangle rectangle = new Rectangle();` `rectangle.setX(150.0f);` `rectangle.setY(75.0f);` `rectangle.setWidth(300.0f);` `rectangle.setHeight(150.0f);` `rectangle.setArcWidth(30.0);` `rectangle.setArcHeight(20.0);`
Circle	`Circle`	`Circle circle = new Circle();` `circle.setCenterX(300.0f);` `circle.setCenterY(135.0f);` `circle.setRadius(100.0f);`

Classes for Shape (`javafx.scene.shape`) (Cont.)

Shape	Class	Example
Ellipse	`Ellipse`	`Ellipse ellipse = new Ellipse();` `ellipse.setCenterX(300.0f);` `ellipse.setCenterY(150.0f);` `ellipse.setRadiusX(150.0f);` `ellipse.setRadiusY(75.0f);`
Polygon	`Polygon`	`Polygon polygon = new Polygon();` `polygon.getPoints().addAll(new Double[]{` `300.0, 50.0,` `450.0, 150.0,` `300.0, 250.0,` `150.0, 150.0,` `});`

Classes for Shape (`javafx.scene.shape`) (Cont.)

Shape	Class	Example
Polyline	Polyline	`Polyline polyline = new Polyline();` `polyline.getPoints().addAll(new Double[]{` `200.0, 50.0,` `400.0, 50.0,` `450.0, 150.0,` `400.0, 250.0,` `200.0, 250.0,` `150.0, 150.0,` `});`
Cubic Curve	CubicCurve	`CubicCurve cubicCurve = new CubicCurve();` `cubicCurve.setStartX(100.0f);` `cubicCurve.setStartY(150.0f);` `cubicCurve.setControlX1(400.0f);` `cubicCurve.setControlY1(40.0f);` `cubicCurve.setControlX2(175.0f);` `cubicCurve.setControlY2(250.0f);` `cubicCurve.setEndX(500.0f);` `cubicCurve.setEndY(150.0f);`

Classes for Shape (`javafx.scene.shape`) (Cont.)

Shape	Class	Description
Quad Curve	QuadCurve	`QuadCurve quadCurve = new QuadCurve();` `quadCurve.setStartX(100.0);` `quadCurve.setStartY(220.0f);` `quadCurve.setEndX(500.0f);` `quadCurve.setEndY(220.0f);` `quadCurve.setControlX(250.0f);` `quadCurve.setControlY(0.0f);`
Arc	Arc	`Arc arc = new Arc();` `arc.setCenterX(100.0);` `arc.setCenterY(100.0);` `arc.setRadiusX(100.0);` `arc.setRadiusY(100.0);` `arc.setStartAngle(0.0);` `arc.setLength(100.0);`

Example (Adding 2-D Shapes)

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.Ellipse;
import javafx.stage.Stage;

public class MyFirstGUI extends Application {
    public void start(Stage primaryState) throws Exception {
        Ellipse ellipse = new Ellipse();
        ellipse.setCenterX(300.0f); ellipse.setCenterY(150.0f);
        ellipse.setRadiusX(150.0f); ellipse.setRadiusY(75.0f);

        Group root = new Group(ellipse);
        Scene s = new Scene(root, 600, 400);
        primaryState.setScene(s);
        primaryState.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Create object of Ellipse class and set its properties.

Pass object of ellipse as argument to Group contructor

JavaFX - Colors

- `javafx.scene.paint` package provides various classes to apply colors to an application. This package contains an abstract class named `Paint` and it is the base class of all the classes that are used to apply colors.
- Using these classes, you can apply colors in the following patterns
  - Uniform – color is applied uniformly throughout node.
  - Image Pattern – fills the region of the node with an image pattern.
  - Gradient – the color applied to the node varies from one point to the other. It has two kinds of gradients namely Linear Gradient and Radial Gradient.
- Instance of `Color` class can be created by providing Red, Green, Blue and Opacity value ranging from 0 to 1 in double.

```
Color color = new Color(double red, double green, double blue, double opacity);
Color color = new Color(0.0,0.3,0.2,1.0);
```

- Instance of `Color` class can be created using following methods also

```
Color c = Color.rgb(0,0,255);          //passing RGB values
Color c = Color.hsb(270,1.0,1.0);      //passing HSB values
Color c = Color.web("0x0000FF",1.0); //passing hex code
```

Applying Color to the Nodes

- `setFill(Color)` method is used to apply color to nodes such as Shape, Text, etc.
- `setStroke(Color)` method is used to apply strokes to the nodes.

```
//Setting color to the text
Color color = new Color.BEIGE
text.setFill(color);

//Setting color to the stroke
Color color = new Color.DARKSLATEBLUE
circle.setStroke(color);
```

Color Example

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.shape.Circle;

public class MyFirstGUI extends Application {
    public void start(Stage stage) {
        Circle circle = new Circle(); circle.setCenterX(300.0f); circle.setCenterY(180.0f);
        circle.setRadius(90.0f);
        circle.setFill(Color.RED); circle.setStrokeWidth(3); circle.setStroke(Color.GREEN);
        Group root = new Group(circle);
        Scene scene = new Scene(root, 600, 300);
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String args[]) {
        launch(args);
    }
}
```

JavaFX – Image

- You can load and modify images using the classes provided by JavaFX in the package `javafx.scene.image`.
- JavaFX supports the image formats like Bmp, Gif, Jpeg, Png.
- Class Image of `javafx.scene.image` package is used to load an image
- Any of the following argument is required to the constructor of the class
  - An InputStream object of the image to be loaded or

```
FileInputStream inputstream = new FileInputStream ("C:\\image.jpg");
Image image = new Image(inputstream);
```

  - A string variable holding the URL for the image.

```
Image image = new Image("http://sample.com/res/flower.png");
```

- After loading image in Image object, view is set to load the image using `ImageView` class

```
ImageView imageView = new ImageView(image);
```

Color Example (Image)

```
import java.io.FileInputStream; import javafx.application.Application;
import javafx.scene.Group; import javafx.scene.Scene; import javafx.scene.image.Image;
import javafx.scene.image.ImageView; import javafx.stage.Stage;

public class ImageExample extends Application {
    public void start(Stage stage) throws Exception {
        Image image = new Image(new FileInputStream("D://CEJDV.jpg"));
        ImageView imageView = new ImageView(image);
        imageView.setX(50);
        imageView.setY(25);
        imageView.setFitHeight(455);
        imageView.setFitWidth(500);
        imageView.setPreserveRatio(true);
        Group root = new Group(imageView);
        Scene scene = new Scene(root, 600, 500);
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String args[]) { launch(args); }
}
```

Layout Panes

- After constructing all the required nodes in a scene, we will generally arrange them in order.
- This arrangement of the components within the container is called the Layout of the container.
- JavaFX provides several predefined layouts such as HBox, VBox, Border Pane, Stack Pane, Text Flow, Anchor Pane, Title Pane, Grid Pane, Flow Panel, etc.
- Each of the above mentioned layout is represented by a class and all these classes belongs to the package `javafx.layout`. The class named `Pane` is the base class of all the layouts in JavaFX.

Layout Panes (`javafx.scene.layout`)

Sr.	Shape & Description
1	HBox • The HBox layout arranges all the nodes in our application in a single horizontal row. • The class named HBox of the package javafx.scene.layout represents the text horizontal box layout.
2	VBox • The VBox layout arranges all the nodes in our application in a single vertical column. • The class named VBox of the package javafx.scene.layout represents the text Vertical box layout.
3	BorderPane • The Border Pane layout arranges the nodes in our application in top, left, right, bottom and center positions. • The class named BorderPane of the package javafx.scene.layout represents the border pane layout.

Layout Panes (`javafx.scene.layout`) (Cont.)

Sr.	Shape & Description
4	StackPane • The stack pane layout arranges the nodes in our application on top of another just like in a stack. The node added first is placed at the bottom of the stack and the next node is placed on top of it. • The class named StackPane of the package javafx.scene.layout represents the stack pane layout.
5	TextFlow • The Text Flow layout arranges multiple text nodes in a single flow. • The class named TextFlow of the package javafx.scene.layout represents the text flow layout.
6	AnchorPane • The Anchor pane layout anchors the nodes in our application at a particular distance from the pane. • The class named AnchorPane of the package javafx.scene.layout represents the Anchor Pane layout.

Layout Panes (`javafx.scene.layout`) (Cont.)

Sr.	Shape & Description
7	TilePane • The Tile Pane layout adds all the nodes of application in the form of uniformly sized tiles. • The class named TilePane of the package javafx.scene.layout represents the TilePane layout.
8	GridPane • The Grid Pane layout arranges the nodes in our application as a grid of rows and columns. This layout comes handy while creating forms. • The class named GridPane of the package javafx.scene.layout represents the GridPane layout.
9	FlowPane • The flow pane layout wraps all the nodes in a flow. A horizontal flow pane wraps the elements of the pane at its height, while a vertical flow pane wraps the elements at its width. • The class named FlowPane of the package javafx.scene.layout represents the Flow Pane layout.

Creating a Layout

- To create a layout, you need to –
  - Create nodes.
  - Instantiate the respective class of the required layout.
  - Set the properties of the layout.
  - Add all the created nodes to the layout.

```
public void start(Stage stage) {
    TextField textField = new TextField();
    Button playButton = new Button("Play");
    Button stopButton = new Button("stop");
    HBox hbox = new HBox();
    hbox.setSpacing(10);
    hbox.setMargin(textField, new Insets(20, 20, 20, 20));
    hbox.setMargin(playButton, new Insets(20, 20, 20, 20));
    hbox.setMargin(stopButton, new Insets(20, 20, 20, 20));
    ObservableList<Node> list = hbox.getChildren();
    list.addAll(textField, playButton, stopButton);
    Scene scene = new Scene(hbox);
    stage.setScene(scene);
    stage.show();
}
```

JavaFX - Events

- In GUI applications, web applications and graphical applications, whenever a user interacts with the application (nodes), an event is said to have been occurred.
- For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.
- JavaFX provides support to handle a wide varieties of events. The class named `Event` of the package `javafx.event` is the base class for an event.
- JavaFX provides a wide variety of events. Some of them are as follows:
  1. Mouse Event – occurs when a mouse is clicked.
     - Class – `MouseEvent`
     - Actions - mouse clicked, mouse pressed, mouse released, mouse moved, mouse entered target, mouse exited target, etc.
  2. Key Event – indicates the key stroke occurred on a node.
     - Class – `KeyEvent`
     - Actions - key pressed, key released and key typed.
  3. Drag Event – occurs when the mouse is dragged.
     - Class - `DragEvent`.
     - Actions - drag entered, drag dropped, drag entered target, drag exited target, drag over, etc.
  4. Window Event – occurs when window showing/hiding takes place.
     - Class – `WindowEvent`
     - Actions – window hiding, window shown, window hidden, window showing, etc.

Event Handling

- Event Handling is the mechanism that controls the event and decides what should happen, if an event occurs. This mechanism has the code which is known as an event handler that is executed when an event occurs.
- JavaFX provides handlers and filters to handle events. In JavaFX every event has
  - *Target* – The node on which an event occurred. A target can be a window, scene, and a node.
  - *Source* – The source from which the event is generated will be the source of the event.
  - *Type* – Type of the occurred event; in case of mouse event – mouse pressed, mouse released are the type of events.
- Phases of Event Handling
  - Target selection
  - Route Construction
  - Event Capturing Phase
  - Event Bubbling Phase

Event Handling Phases

- Target selection
  - When an action occurs, the system determines which node is the target based on internal rules:
  - Key events - the target is the node that has focus.
  - Mouse events - the target is the node at the location of the cursor.
  - Gesture events - the target is the node at the center point of all touches at the beginning of the gesture.
  - Swipe events - the target is the node at the center of the entire path of all of the fingers.
  - Touch events - the target for each touch point is the node at the location first pressed.
- Route Construction
  - Whenever an event is generated, the default/initial route of the event is determined by construction of an *Event Dispatch chain*. It is the path from the stage to the source node.
- Event Capturing Phase
  - After the construction of the event dispatch chain, the root node of the application dispatches the event.
  - This event travels to all nodes in the dispatch chain (from top to bottom).
  - If any of these nodes has a filter registered for the generated event, it will be executed.
  - If none of the nodes in the dispatch chain has a filter for the event generated, then it is passed to the target node and finally the target node processes the event.

Event Handling Phases (Cont.)

- Event Bubbling Phase
  - In the event bubbling phase, the event is travelled from the target node to the stage node (bottom to top).
  - If any of the nodes in the event dispatch chain has a handler registered for the generated event, it will be executed.
  - If none of these nodes have handlers to handle the event, then the event reaches the root node and finally the process will be completed.

Event Handlers and Filters

- Event filters and handlers are those which contains application logic to process an event.
- A node can register to more than one handler/filter. In case of parent–child nodes, you can provide a common filter/handler to the parents, which is processed as default for all the child nodes.
- During the event capturing phase, a filter is executed and during the event bubbling phase, a handler is executed.
- All the handlers and filters implement the interface `EventHandler` of the package `javafx.event`.

Handling Mouse Event

```
public class JavafxSample extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    public void start(Stage primaryStage) {
        Group root = new Group();
        Scene scene =
                new Scene(root, 300, 250);
        scene.setOnMouseClicked(mouseHandler);
        scene.setOnMouseDragged(mouseHandler);
        scene.setOnMouseEntered(mouseHandler);
        scene.setOnMouseExited(mouseHandler);
        scene.setOnMouseMoved(mouseHandler);
        scene.setOnMousePressed(mouseHandler);
        scene.setOnMouseReleased(mouseHandler);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

```
    EventHandler<MouseEvent> mouseHandler =
        new EventHandler<MouseEvent>(){
          @Override
       public void handle(MouseEvent mouseEvent){
          System.out.println(
            mouseEvent.getEventType() + "\n" +
            "X : Y - "
            + mouseEvent.getX() + " : " +
            mouseEvent.getY() + "\n" +
            "SceneX : SceneY - "
            + mouseEvent.getSceneX()+" :
            "+mouseEvent.getSceneY() +
            "\n" + "ScreenX : ScreenY - "
            + mouseEvent.getScreenX()+" :
            "+mouseEvent.getScreenY());
        }
    };
}
```

Creating a Calculator using JavaFX

Animation

- JavaFX provides easy to use animation API (javafx.animation package).
- There are some predefined animation that can be used out of the box or you can implement custom animations using KeyFrames.
- Following are the main predefined animations in JavaFX.
  - **TranslateTransition**: Translate transition allows to create movement animation from one point to another within a duration. Using TranslateTransition#**setByX** / TranslateTransition#**setByY**, you can set how much it should move in x and y axis respectively. It also possible to set precise destination by using TranslateTransition#**setToX** / TranslateTransition#**setToY**.
  - **ScaleTransition**: Scale transition is another JavaFX animation which can be used out of the box that allows to animate the scale / zoom of the given object. The object can be enlarged or minimized using this animation.
  - **RotateTransition**: Rotate transition provides animation for rotating an object. We can provide upto what angle the node should rotate by **toAngle.** Using **byAngle** we can specify how much it should rotate from current angle of rotation.
  - **FadeTransition**: Fade transition creates a fade in / fade out effect by controlling opacity of the object. We can make fade in transition or fade out transition in JavaFX by setting the to and from value.
  - **PathTransition**: Path transition provides option to move object through a specified path. The path can be anything from simple straight line to complex quadratic curves.

Example Animation

```
@Override
public void start(Stage primaryStage) throws Exception {
    // TODO Auto-generated method stub
    Rectangle rect = new Rectangle(200,400);
    rect.setX(400);
    rect.setY(150);

    Group root = new Group(rect);

    Scene s = new Scene(root, 1000,700);

    Duration dt = new Duration(2500);
    RotateTransition rt = new RotateTransition(dt,rect);
    rt.setByAngle(180);
    rt.play();

    primaryStage.setScene(s);
    primaryStage.show();

}
```

Exercise

- Creating a Calculator using JavaFX
- Write a program that displays a tic-tac-toe board. A cell may be X, O, or empty. What to display at each cell is randomly decided. The X and O are images in the files X.gif and O.gif.
- Write a program that displays the color of a circle as red when the mouse button is pressed and as white when the mouse button is released.

Object Oriented Programming -I (3140705)

Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

Unit-08

JavaFX UI Controls & Multimedia

Prof. Jayesh D. Vagadiya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot
Jayesh.vagadiya@darshan.ac.in
9537133260

Outline

- ✓ Label
- ✓ Button
- ✓ CheckBox
- ✓ RadioButton
- ✓ TextField
- ✓ TextArea
- ✓ ComboBox
- ✓ ListView
- ✓ ScrollBar
- ✓ Slider
- ✓ Video
- ✓ Audio

Label

- Label is used to display a short text or an image, it is a non-editable text control.
- It is useful for displaying text that is required to fit within a specific space.
- Label can only display text or image and it cannot get focus.
- Constructor for the Label class are:

Constructor	Description
Label()	Creates an empty Label
Label(String text)	Creates Label with supplied text
Label(String text, Node graphics)	Creates a Label with the supplied text and graphic.

Label Example

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class LabelExample extends Application  {
    @Override
    public void start(Stage primaryStage) throws Exception {

        Label label = new Label("Welcome to JavaFX");

        Scene scene = new Scene(label, 200, 200);
       primaryStage.setTitle("JavaFX Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

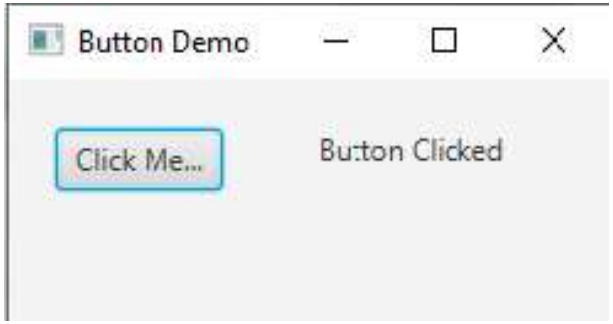
Button

- Button control enables an application to have some action executed when the application user clicks the button.
- The button control can contain text and/or a graphic.
- When a button is pressed and released a ActionEvent is sent. Some action can be performed based on this event by implementing an EventHandler to process the ActionEvent.
- Buttons can also respond to mouse events by implementing an EventHandler to process the MouseEvent.
- Constructor for the Button class are:

Constructor	Description
Button()	Creates a button with an empty string for its label.
Button(String text)	Creates a button with the specified text as its label.
Button(String text, Node graphic)	Creates a button with the specified text and icon for its label.

Button Example

```
@Override
public void start(Stage primaryStage) {
    Button btn = new Button();
    Label lbl = new Label();
    btn.setText("Click Me...");
    btn.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            lbl.setText("Button Clicked");
        }
    });
    HBox root = new HBox();
    root.setMargin(btn, new Insets(20,20,20,20));
    root.setMargin(lbl, new Insets(20,20,20,20));
    root.getChildren().add(btn);
    root.getChildren().add(lbl);
    primaryStage.setTitle("Button Demo");
    primaryStage.setScene(new Scene(root, 250, 100));
    primaryStage.show();
}
```



Checkbox

- The Check Box is used to provide more than one choices to the user.
- It can be used in a scenario where the user is prompted to select more than one option.
- It is different from the radiobutton in the sense that, we can select more than one checkboxes in a scenerio.
- Constructor for the Checkbox class are:

Constructor	Description
CheckBox()	Creates a check box with an empty string for its label.
CheckBox(String text)	Creates a check box with the specified text as its label.

Checkbox Example

```
@Override
public void start(Stage primaryStage) throws Exception {
    Label l = new Label("What do you listen: ");
    CheckBox c1 = new CheckBox("Big FM");
    CheckBox c2 = new CheckBox("Radio Mirchi");
    CheckBox c3 = new CheckBox("Red FM");
    CheckBox c4 = new CheckBox("MY FM");
    HBox root = new HBox();
    root.getChildren().addAll(l,c1,c2,c3,c4);
    root.setSpacing(5);
    Scene scene=new Scene(root,450,100);
    primaryStage.setScene(scene);
    primaryStage.setTitle("CheckBox Example");
    primaryStage.show();
}
```

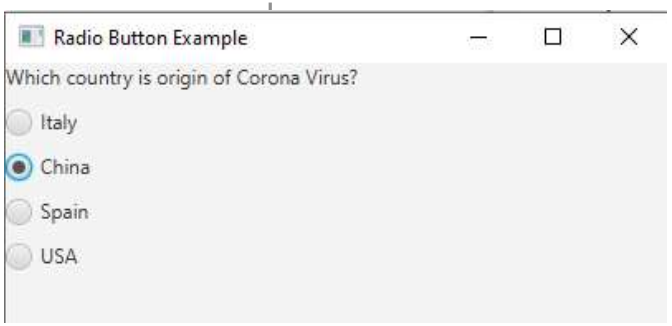
RadioButton

- The Radio Button is used to provide various options to the user.
- The user can only choose one option among all.
- A radio button is either selected or deselected.
- It can be used in a scenario of multiple choice questions in the quiz where only one option needs to be chosen by the student.
- Constructor for the RadioButton class are:

Constructor	Description
RadioButton()	Creates a radio button with an empty string for its label.
RadioButton(String text)	Creates a radio button with the specified text as its label.

Checkbox Example

```
public void start(Stage primaryStage) throws Exception {
    Label lbl = new Label("Which country is origin of Corona Virus?");
    ToggleGroup group = new ToggleGroup();
    RadioButton button1 = new RadioButton("Italy");
    RadioButton button2 = new RadioButton("China");
    RadioButton button3 = new RadioButton("Spain");
    RadioButton button4 = new RadioButton("USA");
    button1.setToggleGroup(group);
    button2.setToggleGroup(group);
    button3.setToggleGroup(group);
    button4.setToggleGroup(group);
    VBox root=new VBox();
    root.setSpacing(10);
    root.getChildren().addAll(lbl,button1,button2,button3,button4);
    Scene scene=new Scene(root,400,300);
    primaryStage.setScene(scene);
    primaryStage.setTitle("Radio Button Example");
    primaryStage.show();
}
```



TextField

- Text input component that allows a user to enter a single line of unformatted text.
- Constructor for the TextField class are:

Constructor	Description
TextField()	Creates a TextField with empty text content.
TextField(String text)	Creates a TextField with initial text content.

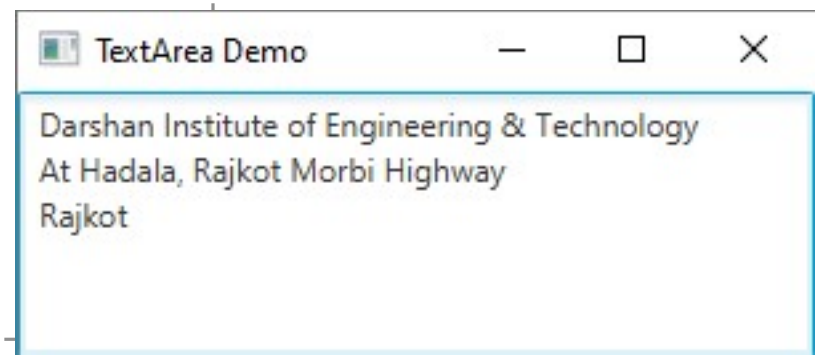
```
Label user_id=new Label("User ID");
Label password = new Label("Password");
TextField tf1=new TextField();
TextField tf2=new TextField();
Button b = new Button("Submit");
GridPane root = new GridPane();
root.addRow(0, user_id, tf1);
root.addRow(1, password, tf2);
root.addRow(2, b);
Scene scene=new Scene(root,300,200);
primaryStage.setScene(scene);
primaryStage.setTitle("Text Field Example");
primaryStage.show();
```

TextArea

- Text input component that allows a user to enter multiple lines of plain text.
- Constructor for the TextArea class are:

Constructor	Description
TextArea()	Creates a TextArea with empty text content.
TextArea(String text)	Creates a TextArea with initial text content.

```
    @Override
    public void start(Stage primaryStage) throws Exception {
        TextArea textArea = new TextArea();
        VBox vbox = new VBox(textArea);
        Scene scene = new Scene(vbox, 300, 100);
        primaryStage.setTitle("TextArea Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```



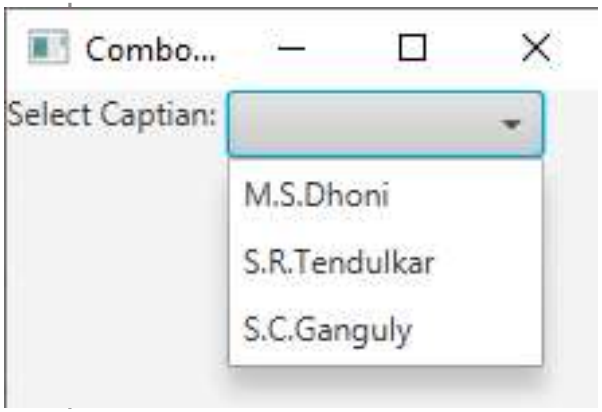
ComboBox

- A combo box is a typical element of a user interface that enables users to choose one of several options.
- A combo box is helpful when the number of items to show exceeds some limit, because it can add scrolling to the drop down list.
- Constructor for the ComboBox class are:

Constructor	Description
ComboBox()	Creates a default ComboBox instance with an empty items list and default selection model.
ComboBox(ObservableList<T> items)	ComboBox(ObservableList<T> items) Creates a default ComboBox instance with the provided items list and a default selection model.

ComboBox Example

```
@Override
public void start(Stage primaryStage) throws Exception {
    Label lbl = new Label("Select Captain: ");
    ObservableList<String> options =
        FXCollections.observableArrayList(
            "M.S.Dhoni",
            "S.R.Tendulkar",
            "S.C.Ganguly"
        );
    ComboBox<String> comboBox = new ComboBox<String>(options);
    HBox hbox = new HBox();
    hbox.getChildren().addAll(lbl,comboBox);
    Scene scene = new Scene(hbox, 200, 120);
    primaryStage.setTitle("ComboBox Experiment 1");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```



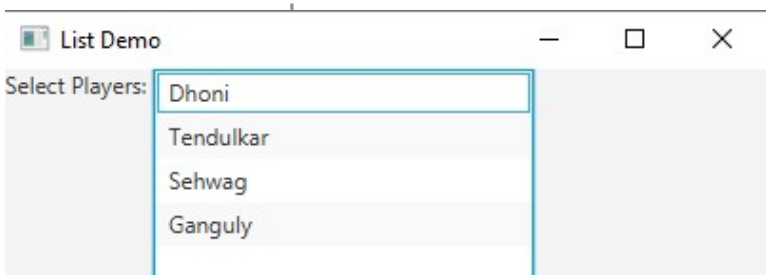
ListView

- A ListView displays a horizontal or vertical list of items from which the user may select, or with which the user may interact.
- Constructor for the ListView class are:

Constructor	Description
ListView()	Creates a default ListView which will display contents stacked vertically.
ListView(ObservableList<T> items)	Creates a default ListView which will stack the contents retrieved from the provided ObservableList vertically.

ListView Example

```
@Override
public void start(Stage primaryStage) throws Exception {
    Label lbl = new Label("Select Players: ");
    ObservableList<String> options =
            FXCollections.observableArrayList(
                "Dhoni","Tendulkar","Sehwag","Ganguly"
            );
    ListView<String> list = new ListView<String>(options);
    list.setPrefSize(200, 50);
    HBox hbox = new HBox();
    hbox.getChildren().addAll(lbl,list);
    Scene scene = new Scene(hbox, 400, 300);
    primaryStage.setTitle("List Demo");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```



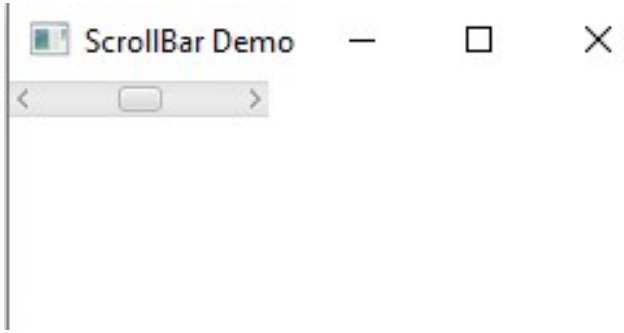
ScrollBar

- JavaFX Scroll Bar is used to provide a scroll bar to the user so that the user can scroll down the application pages.
- Either a horizontal or vertical bar with increment and decrement buttons and a "thumb" with which the user can interact. Typically not used alone but used for building up more complicated controls such as the ScrollPane and ListView.
- Constructor for the ScrollBar class are:

Constructor	Description
ScrollBar()	Creates a new horizontal ScrollBar.

ScrollBar Example

```
    @Override
    public void start(Stage stage) {
        ScrollBar sc = new ScrollBar();
        sc.setMin(0);
        sc.setMax(100);
        sc.setValue(50);
        Group root = new Group();
        Scene scene = new Scene(root, 250, 100);
        stage.setScene(scene);
        root.getChildren().add(sc);
        stage.setTitle("ScrollBar Demo");
        stage.show();
    }
```



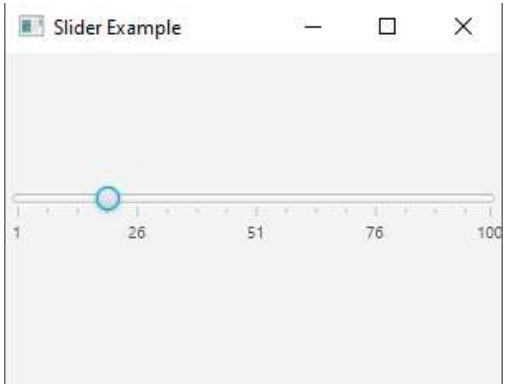
Slider

- The Slider Control is used to display a continuous or discrete range of valid numeric choices and allows the user to interact with the control.
- It is typically represented visually as having a "track" and a "knob" or "thumb" which is dragged within the track. The Slider can optionally show tick marks and labels indicating the different slider position values.
- The three fundamental variables of the slider are min, max, and value. The value should always be a number within the range defined by min and max. min should always be less than or equal to max. min defaults to 0, whereas max defaults to 100.
- Constructor for the Slider class are:

Constructor	Description
Slider()	Creates a default Slider instance.
Slider(double min, double max, double value)	Constructs a Slider control with the specified slider min, max and current value values.

Slider Example

```
    @Override
    public void start(Stage primaryStage) throws Exception {
        Slider slider = new Slider(1,100,20);
        slider.setShowTickLabels(true);
        slider.setShowTickMarks(true);
        StackPane root = new StackPane();
        root.getChildren().add(slider);
        Scene scene = new Scene(root,300,200);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Slider Example");
        primaryStage.show();
    }
```



Video

- In the case of playing video, we need to use the MediaView node to display the video onto the scene.
- For this purpose, we need to instantiate the MediaView class by passing the Mediaplayer object into its constructor. Due to the fact that, MediaView is a JavaFX node, we will be able to apply effects to it.

```
    public void start(Stage primaryStage) throws Exception {
        String path = "G:\\demo.mp4";
        Media media = new Media(new File(path).toURI().toString());
        MediaPlayer mediaPlayer = new MediaPlayer(media);
        MediaView mediaView = new MediaView(mediaPlayer);
        mediaPlayer.setAutoPlay(true);
        Group root = new Group();
        root.getChildren().add(mediaView);
        Scene scene = new Scene(root,1366,768);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Playing video");
        primaryStage.show();
    }
```

Audio

- We can load the audio files with extensions like .mp3,.wav and .aifff by using JavaFX Media API. We can also play the audio in HTTP live streaming format.
- Instantiate javafx.scene.media.Media class by passing the audio file path in its constructor to play the audio files.

```
    public void start (Stage primaryStage) throws Exception {
        String path = "G://demo.mp3";
        Media media = new Media(new File(path).toURI().toString());
        MediaPlayer mediaPlayer = new MediaPlayer(media);
        mediaPlayer.setAutoPlay(true);
        primaryStage.setTitle("Playing Audio");
        primaryStage.show();
    }
```

Object Oriented Programming -I (3140705)

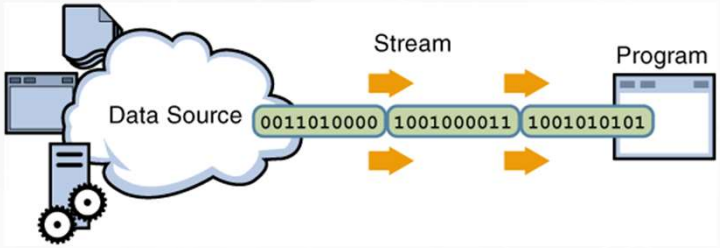
Dedicated Faculty, Committed Education

Darshan

Institute of Engineering & Technology

Unit-09

IO Programming



Prof. Jayesh D. Vagadiya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot
Jayesh.vagadiya@darshan.ac.in
9537133260

Outline

- ✓ File class
- ✓ Stream
- ✓ Byte Stream
- ✓ Character Stream

File class

- Java `File` class represents the files and directory pathnames in an **abstract manner**. This class is used for **creation of files** and **directories**, **file searching**, **file deletion** etc.
- The File object represents the actual file/directory on the disk. Below given is the list of constructors to create a File object.
- Constructors :

Sr.	Constructor
1	File(String pathname) Creates a new File instance by converting the given pathname string into an abstract pathname.
2	File(String parent, String child) Creates a new File instance from a parent pathname string and a child pathname string.
3	File(URI uri) Creates a new File instance by converting the given file: URI into an abstract pathname.

Methods of File Class

Sr.	Method
1	public boolean isAbsolute() Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise
2	public String getAbsolutePath() Returns the absolute pathname string of this abstract pathname.
3	public boolean canRead() Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
4	public boolean canWrite() Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
5	public boolean exists() Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise

Methods of File Class (Cont.)

Sr.	Method
6	public boolean isDirectory() Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.
7	public boolean isFile() Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria
8	public long lastModified() Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970).
9	public long length() Returns the length of the file denoted by this abstract pathname.
10	public boolean delete() Deletes the file or directory.
11	public String[] list() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

File Class Example

```
import java.io.File;
class FileDemo {
    public static void main(String args[]) {
        File f1 = new File("FileDemo.java");
        System.out.println("File Name: " + f1.getName());
        System.out.println("Path: " + f1.getPath());
        System.out.println("Abs Path: " + f1.getAbsolutePath());
        System.out.println("Parent: " + f1.getParent());
        System.out.println(f1.exists() ? "exists" : "does not exist");
        System.out.println(f1.canWrite() ? "is writeable" : "is not writeable");
        System.out.println (f1.canRead () ? "is readable" : "is not readable");
        System.out.println ("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
        System.out.println(f1.isFile() ? "is normal file" : "might be a named pipe");
        System.out.println(f1.isAbsolute() ? "is absolute" : "is not absolute");
        System.out.println("File last modified: " + f1.lastModified());
        System.out.println("File size: " + f1.length() + " Bytes");
    }
}
```

Stream

- A stream can be defined as a sequence of data.
- All streams represent an input source and an output destination.
- There are two kinds of Streams
  - **Byte Stream**
  - **Character Stream**
- The **java.io** package contains all the classes required for input-output operations.
- The stream in the java.io package supports all the datatype including primitive.

Byte Streams

- Byte streams provide a convenient means for handling input and output of bytes.
- Byte streams are used, for example, when reading or writing binary data.

FileOutputStream

- Java `FileOutputStream` is an output stream for writing data to a file.
- `FileOutputStream` will create the file before opening it for output.
- On opening a read only file, it will throw an exception.

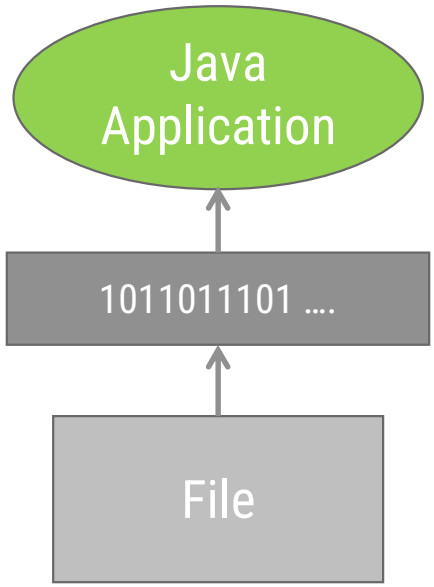
Sr.	Method
1	**void write(byte[] b)** This method writes b.length bytes from the specified byte array to this file output stream.
2	**void write(byte[] b, int off, int len)** This method writes len bytes from the specified byte array starting at offset off to this file output stream.
3	**void write(int b)** This method writes the specified byte to this file output stream.
4	**void close()** This method closes this file output stream and releases any system resources associated with this stream.

FileOutputStream Example

```
class FileOutDemo {
    public static void main(String args[]) {
        try {
            FileOutputStream fout = new FileOutputStream("abc.txt");
            String s = "Sourav Ganguly is my favorite player";
            byte b[] = s.getBytes();
            fout.write(b);
            fout.close();
            System.out.println("Success...");
        } catch (Exception e) {
                System.out.println(e);
        }
    }
}
```

FileInputStream

- `FileInputStream` class is used to read bytes from a file.
- It should be used to read byte-oriented data for example to read image, audio, video etc.



Sr.	Method
1	**public int read()**
2	**public int read(byte[] b)** b - the buffer into which the data is read. Returns: the total number of bytes read into the buffer, or -1.
3	**public int read(byte[] b, int off, int len)** b - the buffer into which the data is read. off - the start offset in the destination array b len - the maximum number of bytes read. Returns: the total number of bytes read into the buffer, or -1
4	**public long skip(long n)** n - the number of bytes to be skipped. Returns: the actual number of bytes skipped.
5	**public int available()** an estimate of the number of remaining bytes that can be read
6	**public void close()** Closes this file input stream and releases any system resources associated.

FileInputStream Example

```
class SimpleRead {
    public static void main(String args[]) {
        try {
            FileInputStream fin = new FileInputStream("abc.txt");
            int i = 0;
            while ((i = fin.read()) != -1) {
                System.out.println((char) i);
            }
            fin.close();
        } catch (Exception e) {
                System.out.println(e);
        }
    }
}
```

Example of Byte Streams

```
import java.io.*;
public class CopyFile {
    public static void main(String args[]) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    } }
```

Character Streams

- Character Streams provide a convenient means for handling input and output of characters.
- Internationalization is possible as it uses Unicode.
- For character streams we have two base classes
  - Reader
  - Writer

Reader

- The Java `Reader` class is the base class of all Reader's in the IO API.
- Subclasses include a `FileReader`, `BufferedReader`, `InputStreamReader`, `StringReader` and several others.
- Here is a simple Java IO Reader example:

```
Reader reader = new FileReader("c:\\data\\myfile.txt");
int data = reader.read();
while (data != -1) {
    char dataChar = (char) data;
    data = reader.read();
}
```

- Combining Readers with InputStream

```
Reader reader = new InputStreamReader("c:\\data\\myfile.txt");
```

Writer

- The Java `Writer` class is the base class of all Writers in the I-O API.
- Subclasses include `BufferedWriter`, `PrintWriter`, `StringWriter` and several others.
- Here is a simple Java IO Writer example:

```
Writer writer = new FileWriter("c:\\data\\file-output.txt");
writer.write("Hello World Writer");
writer.close();
```

- Combining Readers With OutputStreams

```
Writer writer = new OutputStreamWriter("c:\\data\\file-output.txt");
```

BufferedReader

- The `java.io.BufferedReader` class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- Following are the important points about `BufferedReader`:
  - The buffer size may be specified, or the default size may be used.
  - Each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream.
- Constructors :

Sr.	Constructor
1	BufferedReader(Reader in) This creates a buffering character-input stream that uses a default-sized input buffer.
2	BufferedReader(Reader in, int sz) This creates a buffering character-input stream that uses an input buffer of the specified size.

BufferedReader (Methods)

Sr.	Methods
1	void close() This method closes the stream and releases any system resources associated with it.
2	int read() This method reads a single character.
3	int read(char[] cbuf, int off, int len) This method reads characters into a portion of an array.
4	String readLine() This method reads a line of text.
5	void reset() This method resets the stream.
6	long skip(long n) This method skips characters.

BufferedReader – Example

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

class BufferedReaderDemo {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("input.txt");
        BufferedReader br = new BufferedReader(fr);
        char c[] = new char[20];
        br.skip(8);
        if (br.ready()) {
            System.out.println(br.readLine());
            br.read(c);
            for (int i = 0; i < 20; i++) {
                System.out.print(c[i]);
            }
        }
    }
}
```

Object Oriented Programming -I (3140705)

Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

Unit-10 &11

Collection Framework

Prof. Jayesh D. Vagadiya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot
Jayesh.vagadiya@darshan.ac.in
9537133260

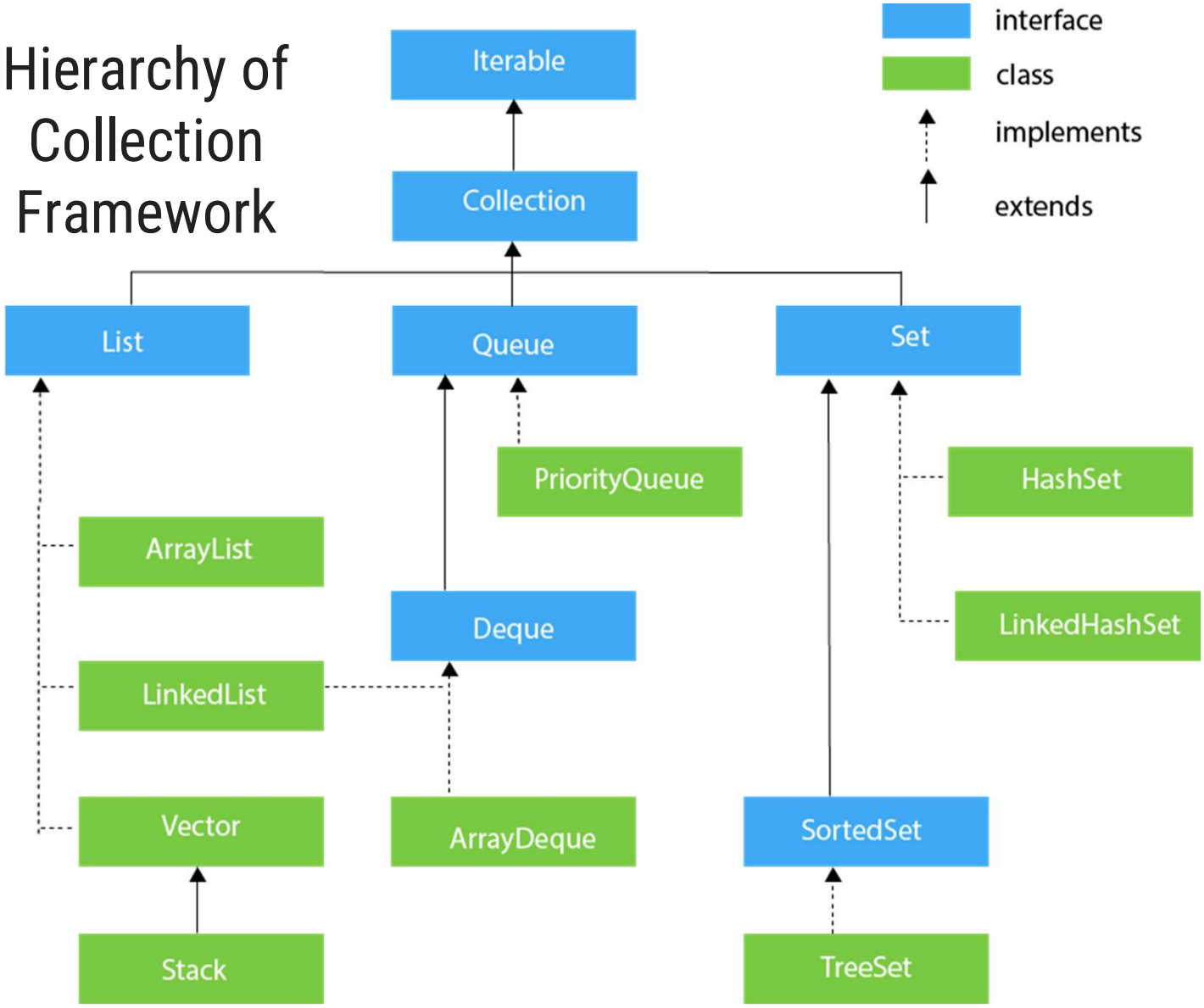
Outline

- ✓ Collection
- ✓ List Interface
- ✓ Iterator
- ✓ Comparator
- ✓ Vector Class
- ✓ Stack
- ✓ Queue
- ✓ List v/s Sets
- ✓ Map Interfaces

Collection

- The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Hierarchy of Collection Framework



Collection Interface - Methods

Sr.	Method & Description
1	boolean add(E e) It is used to insert an element in this collection.
2	boolean addAll(Collection<? extends E> c) It is used to insert the specified collection elements in the invoking collection.
3	void clear() It removes the total number of elements from the collection.
4	booelan contains(Object element) It is used to search an element.
5	boolean containsAll(Collection<?> c) It is used to search the specified collection in the collection.
6	boolean equals(Object obj) Returns true if invoking collection and obj are equal. Otherwise returns false.
7	int hashCode() Returns the hashcode for the invoking collection.

Collection Interface – Methods (Cont.)

Sr.	Method & Description
8	boolean isEmpty() Returns true if the invoking collection is empty. Otherwise returns false.
9	Iterator iterator() It returns an iterator.
10	boolean remove(Object obj) Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.
11	boolean removeAll(Collection<?> c) It is used to delete all the elements of the specified collection from the invoking collection.
12	boolean retainAll(Collection<?> c) It is used to delete all the elements of invoking collection except the specified collection.
13	int size() It returns the total number of elements in the collection.

List Interface

- The **`List`** interface extends **`Collection`** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.
- List is a generic interface with following declaration

```
interface List<E>
```

where `E` specifies the type of object.

List Interface - Methods

Sr.	Method & Description
1	void add(int index, Object obj) Inserts obj into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	boolean addAll(int index, Collection c) Inserts all elements of c into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
3	Object get(int index) Returns the object stored at the specified index within the invoking collection.
4	int indexOf(Object obj) Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.
5	int lastIndexOf(Object obj) Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.

List Interface – Methods (Cont.)

Sr.	Method & Description
6	ListIterator listIterator( ) Returns an iterator to the start of the invoking list.
7	ListIterator listIterator(int index) Returns an iterator to the invoking list that begins at the specified index.
8	Object remove(int index) Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one
9	Object set(int index, Object obj) Assigns obj to the location specified by index within the invoking list.
10	List subList(int start, int end) Returns a list that includes elements from start to end-1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

List Interface (example)

```
import java.util.*;
public class CollectionsDemo {
   public static void main(String[] args) {
      List a1 = new ArrayList();
      a1.add("Sachin");
      a1.add("Sourav");
      a1.add("Shami");
      System.out.println("ArrayList Elements");
      System.out.print("\t" + a1);

      List l1 = new LinkedList();
      l1.add("Mumbai");
      l1.add("Kolkata");
      l1.add("Vadodara");
      System.out.println();
      System.out.println("LinkedList Elements");
      System.out.print("\t" + l1);
   }
}
```

Here **ArrayList** & **LinkedList** implements **List** Interface

```
G:\Darshan\Java 2019\PPTs\HAD\Programs>ja
ArrayList Elements
        [Sachin, Sourav, Shami]
LinkedList Elements
        [Mumbai, Kolkata, Vadodara]
G:\Darshan\Java 2019\PPTs\HAD\Programs>
```

Iterator

- `Iterator` interface is used to cycle through elements in a collection, eg. displaying elements.
- `ListIterator` extends `Iterator` to allow bidirectional traversal of a list, and the modification of elements.
- Each of the collection classes provides an `iterator( )` method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.
- To use an iterator to cycle through the contents of a collection, follow these steps:
  1. Obtain an iterator to the start of the collection by calling the collection's `iterator( )` method.
  2. Set up a loop that makes a call to `hasNext( ).` Have the loop iterate as long as `hasNext( )` returns true.
  3. Within the loop, obtain each element by calling `next( ).`

Iterator - Methods

Sr.	Method & Description
1	boolean hasNext() Returns true if there are more elements. Otherwise, returns false.
2	E next() Returns the next element. Throws NoSuchElementException if there is not a next element.
3	void remove() Removes the current element. Throws IllegalStateException if an attempt is made to call remove() that is not preceded by a call to next()

Iterator - Example

```
import java.util.*;
public class IteratorDemo {
    public static void main(String args[]) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        System.out.print("Contents of list: ");
        Iterator<String> itr = al.iterator();
        while(itr.hasNext()) {
            Object element = itr.next();
            System.out.print(element + " ");
        }
    }
}
```

```
G:\Darshan\Java 2019\PPTs\HAD\Pr
Contents of list: C A E B D F
```

Comparator

- Comparator interface is used to set the sort order of the object to store in the sets and lists.
- The Comparator interface defines two methods: compare( ) and equals( ).
- `int compare(Object obj1, Object obj2)`

  `obj1` and `obj2` are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if `obj1` is greater than `obj2`. Otherwise, a negative value is returned.

- `boolean equals(Object obj)`

  `obj` is the object to be tested for equality. The method returns true if `obj` and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false.

Comparator Example

```
public class ComparatorDemo {
    public static void main(String args[]){
        ArrayList<Student> al=new ArrayList<Student>();
        al.add(new Student("Vijay",23));
        al.add(new Student("Ajay",27));
        al.add(new Student("Jai",21));
        System.out.println("Sorting by age");
        Collections.sort(al,new AgeComparator());
        Iterator<Student> itr2=al.iterator();
        while(itr2.hasNext()){
                Student st=(Student)itr2.next();
                System.out.println(st.name+" "+st.age);
        }
    }
}
```

```
class AgeComparator implements
Comparator<Object>{
    public int compare(Object o1,Object o2){
        Student s1=(Student)o1;
        Student s2=(Student)o2;
        if(s1.age==s2.age) return 0;
        else if(s1.age>s2.age) return 1;
        else return -1;
    }
}
```

```
import java.util.*;
class Student {
    String name;
    int age;
    Student(String name, int
    age){
        this.name = name;
        this.age = age;
    }
}
```

Vector Class

- **`Vector`** implements a dynamic array.
- It is similar to **`ArrayList`**, but with two differences:
  - Vector is synchronized.
  - Vector contains many legacy methods that are not part of the collection framework
- Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.
- Vector is declared as follows:

```
Vector<E> = new Vector<E>;
```

Vector - Constructors

Sr.	Constructor & Description
1	Vector( ) This constructor creates a default vector, which has an initial size of 10
2	Vector(int size) This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size:
3	Vector(int size, int incr) This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward
4	Vector(Collection c) creates a vector that contains the elements of collection c

Vector - Methods

Sr.	Method & Description
7	boolean containsAll(Collection c) Returns true if this Vector contains all of the elements in the specified Collection.
8	Enumeration elements() Returns an enumeration of the components of this vector.
9	Object firstElement() Returns the first component (the item at index 0) of this vector.
10	Object get(int index) Returns the element at the specified position in this Vector.
11	int indexOf(Object elem) Searches for the first occurence of the given argument, testing for equality using the equals method.
12	boolean isEmpty() Tests if this vector has no components.

Vector – Method (Cont.)

Sr.	Method & Description
13	Object lastElement() Returns the last component of the vector.
14	int lastIndexOf(Object elem) Returns the index of the last occurrence of the specified object in this vector.
15	Object remove(int index) Removes the element at the specified position in this Vector.
16	boolean removeAll(Collection c) Removes from this Vector all of its elements that are contained in the specified Collection.
17	Object set(int index, Object element) Replaces the element at the specified position in this Vector with the specified element.
18	int size() Returns the number of components in this vector.

Stack

- `Stack` is a subclass of `Vector` that implements a standard last-in, first-out stack.
- `Stack` only defines the default constructor, which creates an empty stack.
- `Stack` includes all the methods defined by `Vector` and adds several of its own.
- `Stack` is declared as follows:

```
Stack<E> st = new Stack<E>();
```

where `E` specifies the type of object.

Stack - Methods

- Stack includes all the methods defined by Vector and adds several methods of its own.

Sr.	Method & Description
1	boolean empty() Returns true if the stack is empty, and returns false if the stack contains elements.
2	E peek() Returns the element on the top of the stack, but does not remove it.
3	E pop() Returns the element on the top of the stack, removing it in the process.
4	E push(E element) Pushes element onto the stack. Element is also returned.
5	int search(Object element) Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned.

Queue

- `Queue` interface extends `Collection` and declares the behaviour of a queue, which is often a first-in, first-out list.
- `LinkedList` and `PriorityQueue` are the two classes which implements Queue interface
- `Queue` is declared as follows:

```
Queue<E> q = new LinkedList<E>();
Queue<E> q = new PriorityQueue<E>();
```

where `E` specifies the type of object.

Queue - Methods

Sr.	Method & Description
1	E element() Returns the element at the head of the queue. The element is not removed. It throws NoSuchElementException if the queue is empty.
2	boolean offer(E obj) Attempts to add obj to the queue. Returns true if obj was added and false otherwise.
3	E peek() Returns the element at the head of the queue. It returns null if the queue is empty. The element is not removed.
4	E poll() Returns the element at the head of the queue, removing the element in the process. It returns null if the queue is empty.
5	E remove() Returns the element at the head of the queue, returning the element in the process. It throws NoSuchElementException if the queue is empty.

Queue Example

```
import java.util.*;
public class QueueDemo {
    public static void main(String[] args) {
        Queue<String> q = new LinkedList<String>();
        q.add("Tom");
        q.add("Jerry");
        q.add("Mike");
        q.add("Steve");
        q.add("Harry");
        System.out.prin
        System.out.prin
        System.out.prin
        System.out.println("poll(): "+q.poll());
        System.out.println("peek(): "+q.peek());
        System.out.println("Elements in Queue:"+q);
    }
}
```

```
G:\Darshan\Java 2019\PPTs\HAD\Programs>java QueueDemo
Elements in Queue:[Tom, Jerry, Mike, Steve, Harry]
Removed element: Tom
Head: Jerry
poll(): Jerry
peek(): Mike
Elements in Queue:[Mike, Steve, Harry]
```

PriorityQueue

- `PriorityQueue` extends `AbstractQueue` and implements the `Queue` interface.
- It creates a queue that is prioritized based on the queue's comparator.
- `PriorityQueue` is declared as follows:

```
PriorityQueue<E> = new PriorityQueue<E>;
```

- It builds an empty queue with starting capacity as 11.

PriorityQueue - Example

```
import java.util.*;
public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> numbers = new
PriorityQueue<>();
        numbers.add(750);
        numbers.add(500);
        numbers.add(900);
        numbers.add(100);
        while (!numbers.isEmpty()) {
            System.out.println(numbers.remove());
        }
    }
}
```

```
G:\Darshan\Java
100
500
750
900
```

List v/s Sets

List	Set
Lists allow duplicates.	Sets allow only unique elements.
List is an ordered collection.	Sets is an unordered collection.
Popular implementation of List interface includes ArrayList, Vector and LinkedList.	Popular implementation of Set interface includes HashSet, TreeSet and LinkedHashSet.

When to use List and Set?

Lists - If insertion order is maintained during insertion and allows duplicates.

Sets – If unique collection without any duplicates without maintaining order.

Maps

- A map is an object that stores associations between keys and values, or key/value pairs.
- Given a key, you can find its value. Both keys and values are objects.
- The keys must be unique, but the values may be duplicated. Some maps can accept a null key and null values, others cannot.
- Maps don't implement the Iterable interface. This means that you cannot cycle through a map using a for-each style for loop. Furthermore, you can’t obtain an iterator to a map.

Map Interfaces

Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of Map.
NavigableMap	Extends SortedMap to handle the retrieval of entries based on closest-match searches.
SortedMap	Extends Map so that the keys are maintained in ascending order.

Map Classes

Class	Description
AbstractMap	Implements most of the Map interface.
EnumMap	Extends AbstractMap for use with enum keys.
HashMap	Extends AbstractMap to use a hash table.
TreeMap	Extends AbstractMap to use a tree.
WeakHashMap	Extends AbstractMap to use a hash table with weak keys.
LinkedHashMap	Extends HashMap to allow insertion-order iterators.
IdentityHashMap	Extends AbstractMap and uses reference equality when comparing documents.

HashMap Class

- The HashMap class extends AbstractMap and implements the Map interface.
- It uses a hash table to store the map. This allows the execution time of get() and put() to remain constant even for large sets.
- HashMap is a generic class that has declaration:

```
class HashMap<K,V>
```

HashMap - Constructors

Sr.	Constructor & Description
1	HashMap() Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
2	HashMap(int initialCapacity) Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).
3	HashMap(int initialCapacity, float loadFactor) Constructs an empty HashMap with the specified initial capacity and load factor.
4	HashMap(Map<? extends K,? extends V> m) Constructs a new HashMap with the same mappings as the specified Map.

```java
import java.util.*;
class HashMapDemo {
    public static void main(String args[]) {
        // Create a hash map.
        HashMap<String, Double> hm = new HashMap<String, Double>();
        // Put elements to the map
        hm.put("John Doe", new Double(3434.34));
        hm.put("Tom Smith", new Double(123.22));
        hm.put("Jane Baker", new Double(1378.00));
        hm.put("Tod Hall", new Double(99.22));
        hm.put("Ralph Smith", new Double(-19.08));
        // Get a set of the entries.
        Set<Map.Entry<String, Double>> set = hm.entrySet();
        // Display the set.
        for(Map.Entry<String, Double> me : set) {
                System.out.print(me.getKey() + ": ");
                System.out.println(me.getValue());
        }
        System.out.println();
        //Deposit 1000 into John Doe's account.
        double balance = hm.get("John Doe");
        hm.put("John Doe", balance + 1000);
        System.out.println("John Doe's new balance: " +
        hm.get("John Doe"));
    }
}
```

```
C:\Users\hardi\Desktop>java HashMapDemo
Tod Hall: 99.22
John Doe: 3434.34
Ralph Smith: -19.08
Tom Smith: 123.22
Jane Baker: 1378.0

John Doe's new balance: 4434.34
```

Object Oriented Programming -I (3140705)

Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

Unit-12

Concurrency / Multithreading

Prof. Jayesh D. Vagadiya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot
Jayesh.vagadiya@darshan.ac.in
9537133260

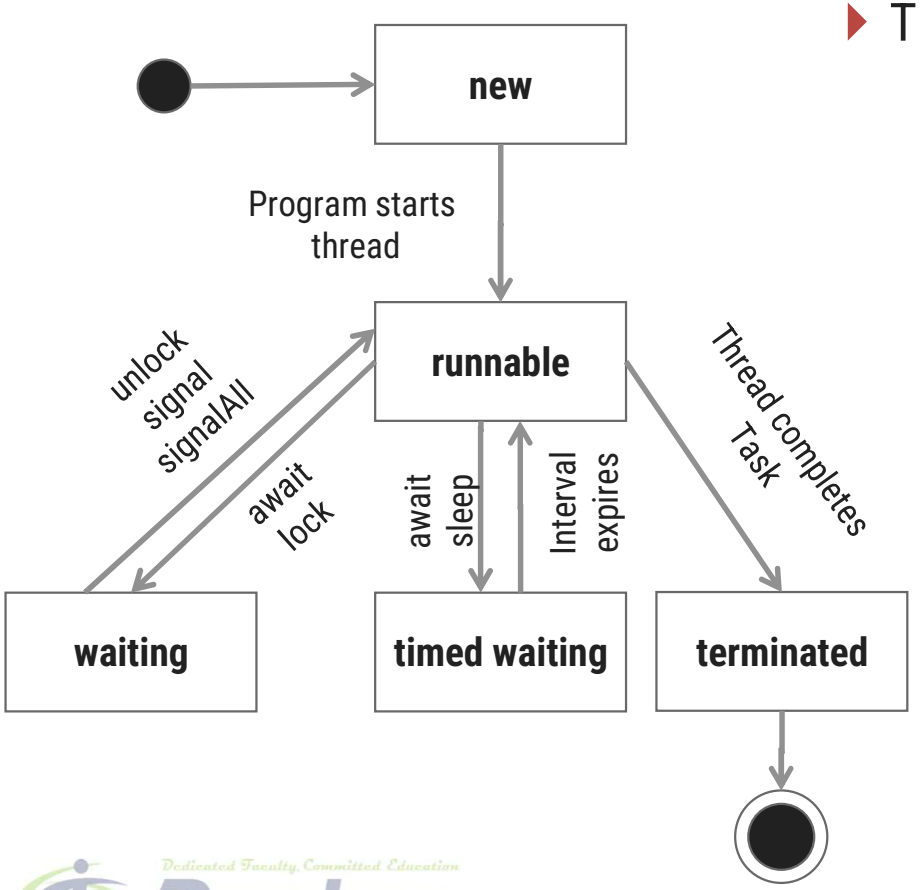
Outline

- ✓ Multithreading
- ✓ Life Cycle of Thread
- ✓ Creating a Thread
  - ✓ Using Thread Class
  - ✓ Using Runnable Interface
- ✓ Join
- ✓ Synchronized

What is Multithreading?

- Multithreading in Java is a process of *executing multiple threads* simultaneously.
- A thread is a *lightweight sub-process*, the smallest unit of processing.
- Multiprocessing and multithreading, both are used to achieve multitasking.
- Threads use a *shared memory area*. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies.
- Lets see the life cycle of the thread.

Life cycle of a Thread



- There are **5** stages in the life cycle of the Thread
  - **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
  - **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
  - **Waiting:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals waiting thread to continue.
  - **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
  - **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Creating a Thread in Java

- There are two ways to create a Thread
  1. **extending** the **Thread class**
  2. **implementing** the **Runnable interface**

1) Extending Thread Class

- One way to create a thread is to create a new class that extends `Thread`, and then to create an instance of that class.
- The extending class must override the `run( )` method, which is the entry point for the new thread.
- It must also call `start( )` to begin execution of the new thread.

```
class NewThread extends Thread {
     NewThread() {
          super("Demo Thread");
          System.out.println("Child thread: " + this);
          start(); // Start the thread
     }
     public void run() {
          try {
               for (int i = 5; i > 0; i--) {
                    System.out.println("Child Thread: " + i);
                    Thread.sleep(500);
               }
          } catch (InterruptedException e) {
              System.out.println("Child interrupted.");
          }
          System.out.println("Exiting child thread.");
     }}
```

```
class ExtendThread {
     public static void main(String args[]) {
          new NewThread(); // create a new thread
          try {
               for (int i = 5; i > 0; i--) {
                    System.out.println("Main Thread: " + i);
                    Thread.sleep(1000);
               }
          } catch (InterruptedException e) {
               System.out.println("Main thread
          interrupted.");
          }
          System.out.println("Main thread exiting.");
     }
}
```

2) Implementing Runnable Interface

- To implement thread using `Runnable` interface, `Runnable` interface needs to be implemented by the class.

```
class NewThread implements Runnable
```

- Class which implements `Runnable` interface should override the `run()` method which containts the logic of the thread.

```
public void run( )
```

- Instance of `Thread` class is created using following constructor.

```
Thread(Runnable threadOb, String threadName);
```

- Here `threadOb` is an instance of a class that implements the `Runnable` interface and the name of the new thread is specified by `threadName`.
- `start()` method of Thread class will invoke the `run()` method.

Example Runnable Interface

Thread using Executor Framework

- Steps to execute thread using Executor Framework are as follows:

1. Create a task (Runnable Object) to execute
2. Create Executor Pool using Executors
3. Pass tasks to Executor Pool
4. Shutdown the Executor Pool

Example Executable Framework

```
class Task implements Runnable {
     private String name;
     public Task(String s) {
          name = s;
     }
     public void run() {
          try {
               for (int i = 1; i<=5; i++) {
                    System.out.println(name+
                    " - task number - "+i);
                    Thread.sleep(1000);
               }
               System.out.println(name+"
               complete");
          }
          catch(InterruptedException e) {
               e.printStackTrace();
          }
     }
}
```

```
import java.util.concurrent.*;

public class ExecutorThreadDemo {
     public static void main(String[] args)
     {
          Runnable r1 = new Task("task 1");
          Runnable r2 = new Task("task 2");
          Runnable r3 = new Task("task 3");
          Runnable r4 = new Task("task 4");
          Runnable r5 = new Task("task 5");
          ExecutorService pool =
          Executors.newFixedThreadPool(3);
          pool.execute(r1);
          pool.execute(r2);
          pool.execute(r3);
          pool.execute(r4);
          pool.execute(r5);
          pool.shutdown();
     }
}
```

```
task 1 - task number - 1
task 2 - task number - 1
task 3 - task number - 1
task 1 - task number - 2
task 2 - task number - 2
task 3 - task number - 2
task 2 - task number - 3
task 1 - task number - 3
task 3 - task number - 3
task 1 - task number - 4
task 2 - task number - 4
task 3 - task number - 4
task 1 - task number - 5
task 2 - task number - 5
task 3 - task number - 5
task 2 complete
task 1 complete
task 4 - task number - 1
task 3 complete
task 5 - task number - 1
task 4 - task number - 2
task 5 - task number - 2
task 4 - task number - 3
task 5 - task number - 3
task 4 - task number - 4
task 5 - task number - 4
task 4 - task number - 5
task 5 - task number - 5
task 4 complete
task 5 complete
```

Thread Synchronization

- When we start *two or more threads* within a program, there may be a situation when multiple threads try to *access the same resource* and finally they can produce unforeseen result due to concurrency issues.
- For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.
- So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time.
- Java programming language provides a very handy way of creating threads and synchronizing their task by using *synchronized methods & synchronized blocks.*

Problem without synchronization (Example)

```
class Table {
  void printTable(int n) {
    for (int i = 1; i <= 5; i++) {
      System.out.print(n * i + " ");
      try {
        Thread.sleep(400);
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }
}
```

```
class MyThread1 extends Thread {
  Table t;
  MyThread1(Table t) {
    this.t = t;
  }
  public void run() {
    t.printTable(5);
  }
}
```

```
C:\WINDOWS\system32\cmd.exe
D:\DegreeDemo\PPTDemo>javac TestSynchronization.java
D:\DegreeDemo\PPTDemo>java TestSynchronization
5 100 200 10 300 15 400 20 500 25
```

```
class MyThread2 extends Thread {
  Table t;
  MyThread2(Table t) {
    this.t = t;
  }
  public void run() {
    t.printTable(100);
  }
}
```

```
public class TestSynchronization {
  public static void main(String args[]){
    Table obj = new Table();
    MyThread1 t1 = new MyThread1(obj);
    MyThread2 t2 = new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```

Solution with synchronized method

```
class Table {
 synchronized void printTable(int n) {
    for (int i = 1; i <= 5; i++) {
      System.out.print(n * i + " ");
      try {
        Thread.sleep(400);
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }
}
```

```
class MyThread1 extends Thread {
  Table t;
  MyThread1(Table t) {
    this.t = t;
  }
  public void run() {
    t.printTable(5);
  }
}
```

```
C:\WINDOWS\system32\cmd.exe
D:\DegreeDemo\PPTDemo>javac TestSynchronization.java
D:\DegreeDemo\PPTDemo>java TestSynchronization
5 10 15 20 25 100 200 300 400 500
```

```
class MyThread2 extends Thread {
  Table t;
  MyThread2(Table t) {
    this.t = t;
  }
  public void run() {
    t.printTable(100);
  }
}
```

```
public class TestSynchronization {
  public static void main(String args[]){
    Table obj = new Table();
    MyThread1 t1 = new MyThread1(obj);
    MyThread2 t2 = new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```

Solution with synchronized blocks

```
class Table {
  void printTable(int n) {
    for (int i = 1; i <= 5; i++) {
      System.out.print(n * i + " ");
      try {
        Thread.sleep(400);
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }
}
```

```
class MyThread1 extends Thread {
  Table t;
  MyThread1(Table t) {
    this.t = t;
  }
  public void run() {
    synchronized (t) {
      t.printTable(5);
    }
  }
```

```
class MyThread2 extends Thread {
  Table t;
  MyThread1(Table t) {
    this.t = t;
  }
  public void run() {
    synchronized (t) {
      t.printTable(100);
    }
  }
}
```

```
public class TestSynchronization {
  public static void main(String args[]){
    Table obj = new Table();
    MyThread1 t1 = new MyThread1(obj);
    MyThread2 t2 = new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```

```
C:\WINDOWS\system32\cmd.exe
D:\DegreeDemo\PPTDemo>javac TestSynchronization.java
D:\DegreeDemo\PPTDemo>java TestSynchronization
5 10 15 20 25 100 200 300 400 500
```