ST. JOSEPH'S
COLLEGE OF ENGINEERING
AND TECHNOLOGY,
- PALAI -

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CST 205 - Object Oriented Programming Using Java

Prof. Sarju S

16 September 2020

Module 2: Core Java Fundamentals

Primitive Data types - Integers, Floating Point Types, Characters, Boolean. Literals, Type Conversion and Casting, Variables, Arrays, Strings, Vector class.

Operators - Arithmetic Operators, Bitwise Operators, Relational Operators, Boolean Logical Operators, Assignment Operator, Conditional (Ternary) Operator, Operator Precedence.

Control Statements - Selection Statements, Iteration Statements and Jump Statements.

Object Oriented Programming in Java - Class Fundamentals, Declaring Objects, Object Reference, Introduction to Methods, Constructors, this Keyword, Method Overloading, Using Objects as Parameters, Returning Objects, Recursion, Access Control, Static Members, Final Variables, Inner Classes, Command Line Arguments, Variable Length Arguments.

Inheritance - Super Class, Sub Class, The Keyword super, protected Members, Calling Order of Constructors, Method Overriding, the Object class, Abstract Classes and Methods, using final with Inheritance.

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Primitive Data types

My First Java Program

```java
/**
* The HelloJava program implements an application that
* simply displays "Hello Java!" to the standard output.
*
* @author  Sarju S
* @version 1.0
* @since   2020-09-15
*/
package basicPrograms;

public class HelloJava {
    public static void main(String[] args) {
        // Prints Hello, World! on standard output.
        System.out.println("Hello World!");
    }
}
```

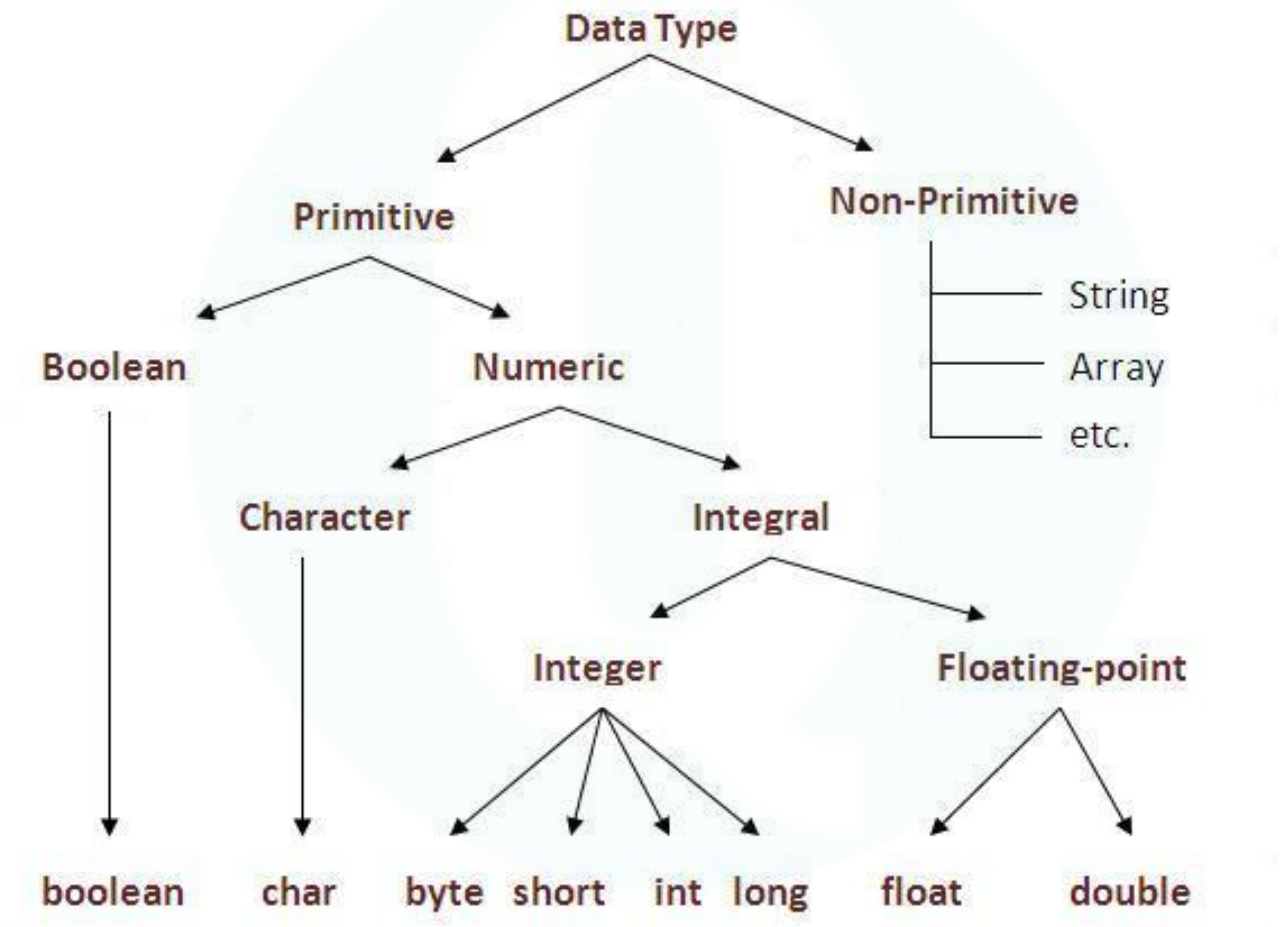

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Data Types

- Variables are the reserved memory locations to store values.
- Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.
- A variable's data type determines the values it may contain, plus the operations that may be performed on it.
- The Java programming language is statically-typed, which means that all variables must first be declared before they can be used.

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Data Types in Java



Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Primitive Data Types

- A primitive type is predefined by the language and is named by a reserved keyword.
- Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean.
- **Integers** : byte, short, int, and long, which are for whole-valued signed numbers.
- **Floating-point numbers** : float and double, which represent numbers with fractional precision.
- **Characters** :char, which represents symbols in a character set, like letters and numbers.
- **Boolean** :boolean, which is a special type for representing true/false values.

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Primitive Data Types - Integer

- Java defines four integer types: byte, short, int, and long.

Name	Width
long	64 bit
int	32 bit
short	16 bit
Byte	8 bit

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Primitive Data Types - Integer

- byte: The smallest integer type is byte.
- This is a signed 8-bit. Variables of type byte are especially useful
  - when you're working with a stream of data from a network or file.
  - when you're working with raw binary data that may not be directly compatible with Java's other built-in types.
- Byte variables are declared by use of the byte keyword.
  - For example, the following declares two byte variables called b and c:

```
byte b, c;
```


Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Primitive Data Types - Integer

- short is a signed 16-bit type.
- you can use a short to save memory in large arrays, in situations where the memory savings actually matters.
  - Example : short s;
- The most commonly used integer type is int.
- In addition to other uses, variables of type int are commonly employed to control loops and to index arrays.
  - Example: int a;
- long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value.
  - Example: long a;

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Primitive Data Types – Floating Point

- Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision.
- The type float specifies a single-precision value that uses 32 bits of storage.
  - Variables of type float are useful when you need a fractional component, but don't require a large degree of precision.
  - Example: float highTemp, lowTemp;
- Double precision, as denoted by the double keyword, uses 64 bits to store a value.
  - When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.
  - Example : double pi, r, a;

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Primitive Data Types – Characters

- In Java, the data type used to store characters is char.
- Java uses Unicode to represent characters
- At the time of Java's creation, Unicode required 16 bits. Thus, in Java char is a 16-bit type.
- Example: char letterA = 'A';

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Primitive Data Types – Booleans

- Java has a primitive type, called boolean, for logical values.
- It can have only one of two possible values, true or false.
- This is the type returned by all relational operators, as in the case of a < b.
  - Example: boolean b;

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Primitive Data Types

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Literals

Literals

- You may have noticed that the new keyword isn't used when initializing a variable of a primitive type.
- Primitive types are special data types built into the language; they are not objects created from a class.
- A literal is the source code representation of a fixed value; literals are represented directly in your code without requiring computation.
- As shown below, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```


Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Integer Literals

- Integer literal are used to represent a decimal, binary, or hexadecimal value.
- Integer literals are used to initialize variables of integer data types byte, short, int and long.
- Integer literal that ends with l or L is of type long.
- In Java, a binary literals starts with 0b, and hexadecimal literals starts with 0x and rest every integer literal is a decimal literals.
  - `int decInt = 18;  // 18 in decimal notation`
  - `int binInt = 0b10010;  // 18 in binary notation`
  - `int hexInt = 0x12;  // 18 in hexadecimal notation`
  - `long longValue = 123456789L;`

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Floating-Point Literals

- A floating-point literal is of type float if it ends with the letter F or f
- Otherwise its type is double and it can optionally end with the letter D or d.
- The floating point types (float and double) can also be expressed using E or e (for scientific notation)

```
double myDouble = 123.4;
double myDoubleScientific = 1.234e2; // same value as d1, but in scientific notation
float myFloat  = 123.4f;
```


Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Character and String Literals

- Literals of types char and String may contain any Unicode (UTF-16) characters.
- String literals are represented as a sequence of characters surrounded by double quotes
- character literal is represented as a single character surrounded by single quotes.
- Java also allows use of escape sequences in string and character literals.
  - For example, \b (backspace), \t (tab), \n (line feed), \f (form feed), \r (carriage return), \" (double quote), \' (single quote), and \\ (backslash).
- `char myChar = 'A'; // Character Literal`
- `char newLine = '\n'; // Character Literal Escape sequence`
- `String myString = "Java Tutorial"; // String Literal`

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Boolean Literals

- Boolean literals are used to represent true or false values
- Example
  - boolean flag = true;

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Using Underscore Characters in Numeric Literals

- In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal.
- This feature enables you, for example. to separate groups of digits in numeric literals, which can improve the readability of your code
- The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

Using Underscore Characters in Numeric Literals

- You can place underscores only between digits; you cannot place underscores in the following places:
  - At the beginning or end of a number
  - Adjacent to a decimal point in a floating point literal
  - Prior to an F or L suffix
  - In positions where a string of digits is expected

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Valid and invalid underscore placements in Numeric Literals

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 =
999_99_9999_L;

// OK (decimal literal)
int x1 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;
// OK (decimal literal)
int x3 = 5_______2;
```

```
// Invalid: cannot put underscores
// in the 0x radix prefix
int x4 = 0_x52;
// Invalid: cannot put underscores
// at the beginning of a number
int x5 = 0x_52;
// OK (hexadecimal literal)
int x6 = 0x5_2;
// Invalid: cannot put underscores
// at the end of a number
int x7 = 0x52_;
```

Variables

Variables

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optional initializer.
- In addition, all variables have a scope, which defines their visibility, and a lifetime.
- Here are several examples of variable declarations of various types. Note that some include an initialization

```
int a, b, c; // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three more ints, initializing d and f.
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x'; // the variable x has the value 'x'.
```


Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

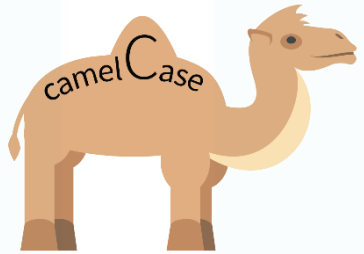
Naming

- Variable names are case-sensitive. A variable's name can be any legal identifier
- The only allowed characters for identifiers are all alphanumeric characters([A-Z],[a-z],[0-9]), '$'(dollar sign) and '_' (underscore).
  - For example "geek@" is not a valid java identifier as it contain '@' special character.
- Identifiers should not start with digits([0-9]). For example "123geeks" is a not a valid java identifier.
- Reserved Words can't be used as an identifier.
  - For example "**int while = 20;**" is an invalid statement as while is a reserved word. There are 53 reserved words in Java.

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Naming

- If the name you choose consists of only one word, spell that word in all lowercase letters.
- If it consists of more than one word, capitalize the first letter of each subsequent word.
  - Examples
    - `gearRatio, currentGear`



- If your variable stores a constant value, such as static final `int NUM_GEARS = 6`, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character.

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

The Scope and Lifetime of Variables

```java
/**
* The ScopeDemo program implements an application that
* simply demonstrates scope of the variables in Java Prograı
*
* @author  Sarju S
* @version 1.0
* @since   2020-09-16
*/
package basicPrograms;

public class ScopeDemo {

    public static void main(String[] args) {
        int x; // known to all code within main
        x = 10;
        if(x == 10) { // start new scope
            int y = 20; // known only to this block
            // x and y both known here.
            System.out.println("x and y: " + x + " " + y);
            x = y * 2;
        }
        //y = 100; // Error! y not known here
        // x is still known here.
        System.out.println("x is " + x);
    }
}
```

Type Conversion and Casting

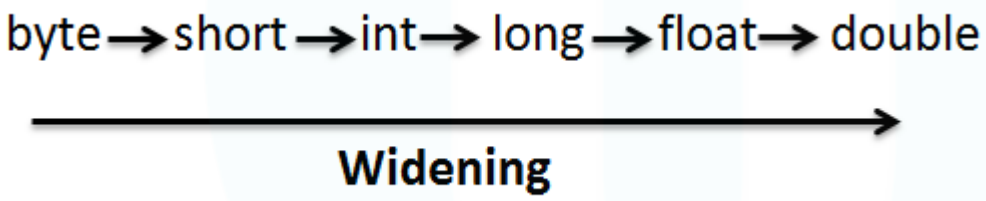
Java Type Conversion

- When a data type is converted into another type is called type conversion (casting).
- When a variable is converted into a different type, the compiler basically treats the variable as of the new data type.
- Type of Type Conversion In Java
  - Implicit Conversion
  - Explicit Conversion

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Implicit Conversion

- implicit or automatic conversion compiler will automatically change one type of data into another.
- Implicit or automatic conversion can happen if both type are compatible and target type is larger than source type.

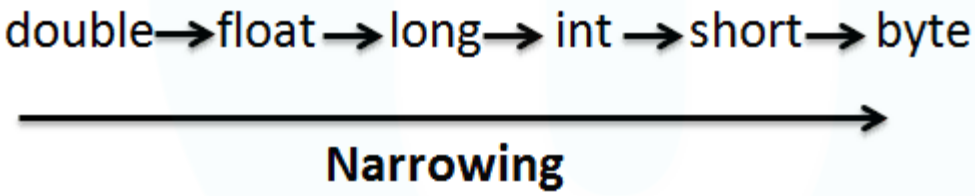


- If you assign an integer value to a floating-point variable, the compiler will insert code to convert the int to a float.
- In implicit conversion a smaller type is converted into a larger thus it is also known as widening conversion.

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Explicit Conversion

- There may be situations when we want to convert a value having larger type to a smaller type.
- In this case casting needs to be performed explicitly.
- Explicit casting allows you to make this type conversion explicit, or to force it when it wouldn't normally happen.



Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Explicit Conversion

- Syntax

```
(type) expression
```

- To perform type casting, put the desired type including modifiers inside parentheses to the left of the variable or constant you want to cast.
- Example

```
int a =10
byte b = (byte) a;
```

- Here, size of source type `int` is 32 bits and size of destination type `byte` is 8 bits.
- Since we are converting a source type having larger size into a destination type having less size, such conversion is known as narrowing conversion.

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

References

- https://docs.oracle.com/javase/tutorial/java
- http://www.java2s.com/Tutorial/Java/0040__Data-Type/LongIntegerLiteral.htm
- https://www.w3adda.com/java-tutorial/java-literals

Thank You

Prof. Sarju S
Department of Computer Science and Engineering
St. Joseph's College of Engineering and Technology, Palai
sarju.s@sjcetpalai.ac.in

CST 205 Object Oriented Programming using Java

CST 281 Object Oriented Programming
(As per KTU 2019 Syllabus)

Module 2 Lecture 1

Module 2 - Core Java Fundamentals

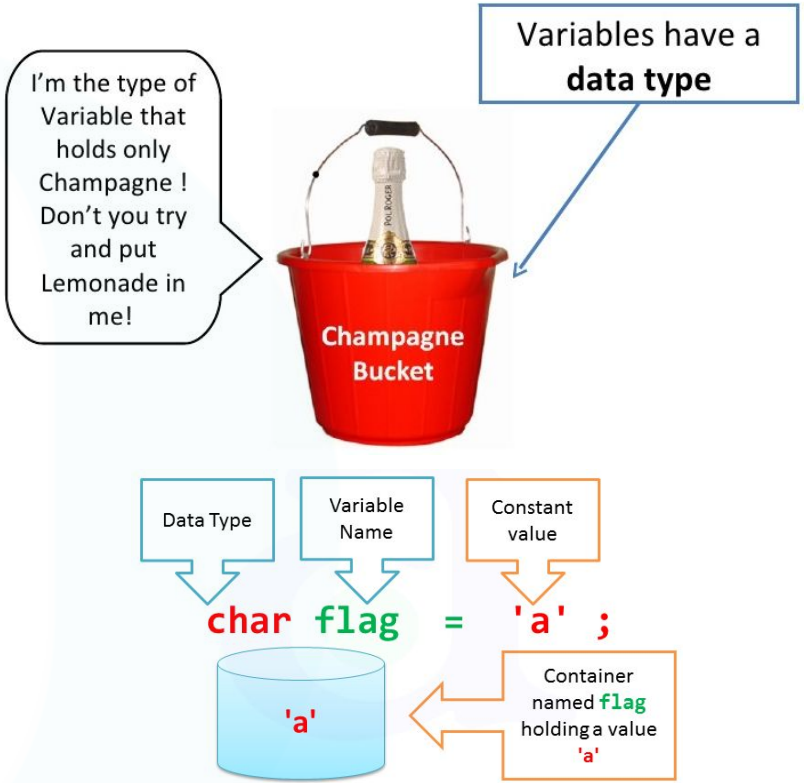
Primitive Data types - Integers, Floating Point Types, Characters, Boolean. Literals, Type Conversion and Casting, Variables, Arrays, Strings, Vector class. **Operators** - Arithmetic Operators, Bitwise Operators, Relational Operators, Boolean Logical Operators, Assignment Operator, Conditional (Ternary) Operator, Operator Precedence. **Control Statements** - Selection Statements, Iteration Statements and Jump Statements.

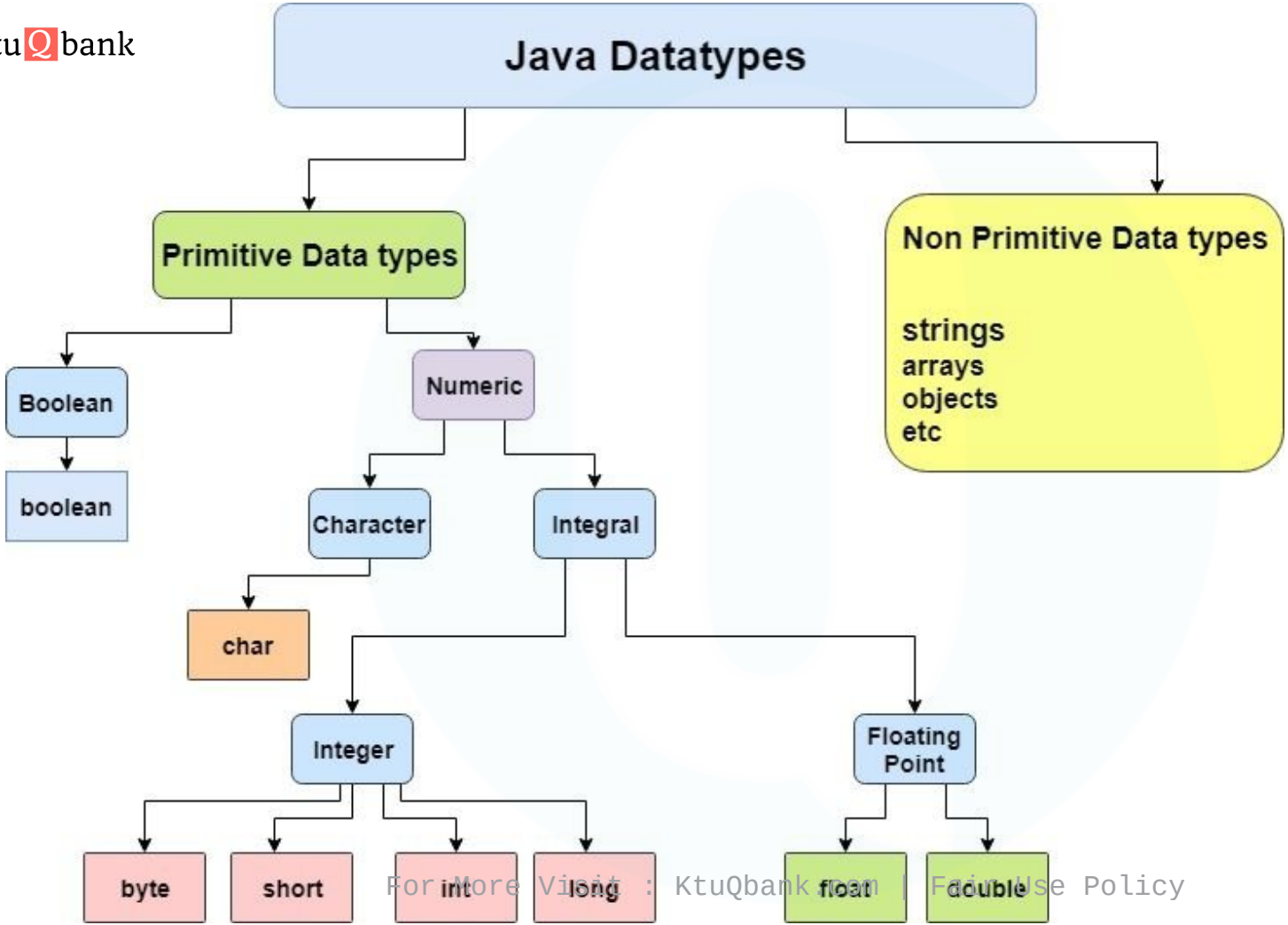
Object Oriented Programming in Java - Class Fundamentals, Declaring Objects, Object Reference, Introduction to Methods, Constructors, this Keyword, Method Overloading, Using Objects as Parameters, Returning Objects, Recursion, Access Control, Static Members, Final Variables, Inner Classes, Command Line Arguments, Variable Length Arguments.

Inheritance - Super Class, Sub Class, The Keyword super, protected Members, Calling Order of Constructors, Method Overriding, the Object class, Abstract Classes and Methods, using final with Inheritance.

Data type

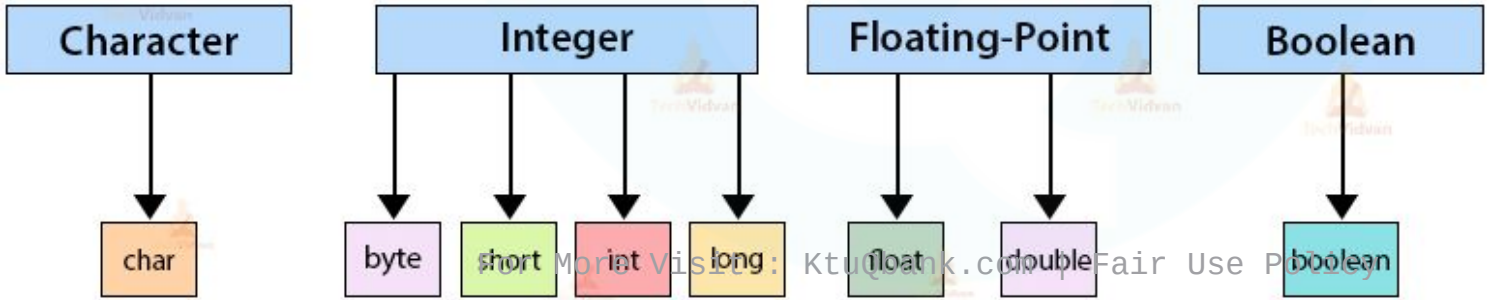
- Data type defines the type of **values that a variable can hold**
- Java is **statically typed**, which means that all variables must first be declared before they can be used.
- Java is **strongly typed**, every variable/expression has a strictly defined data type.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible.
- Any type **mismatches are errors** that must be corrected by the programmer.





Primitive data type

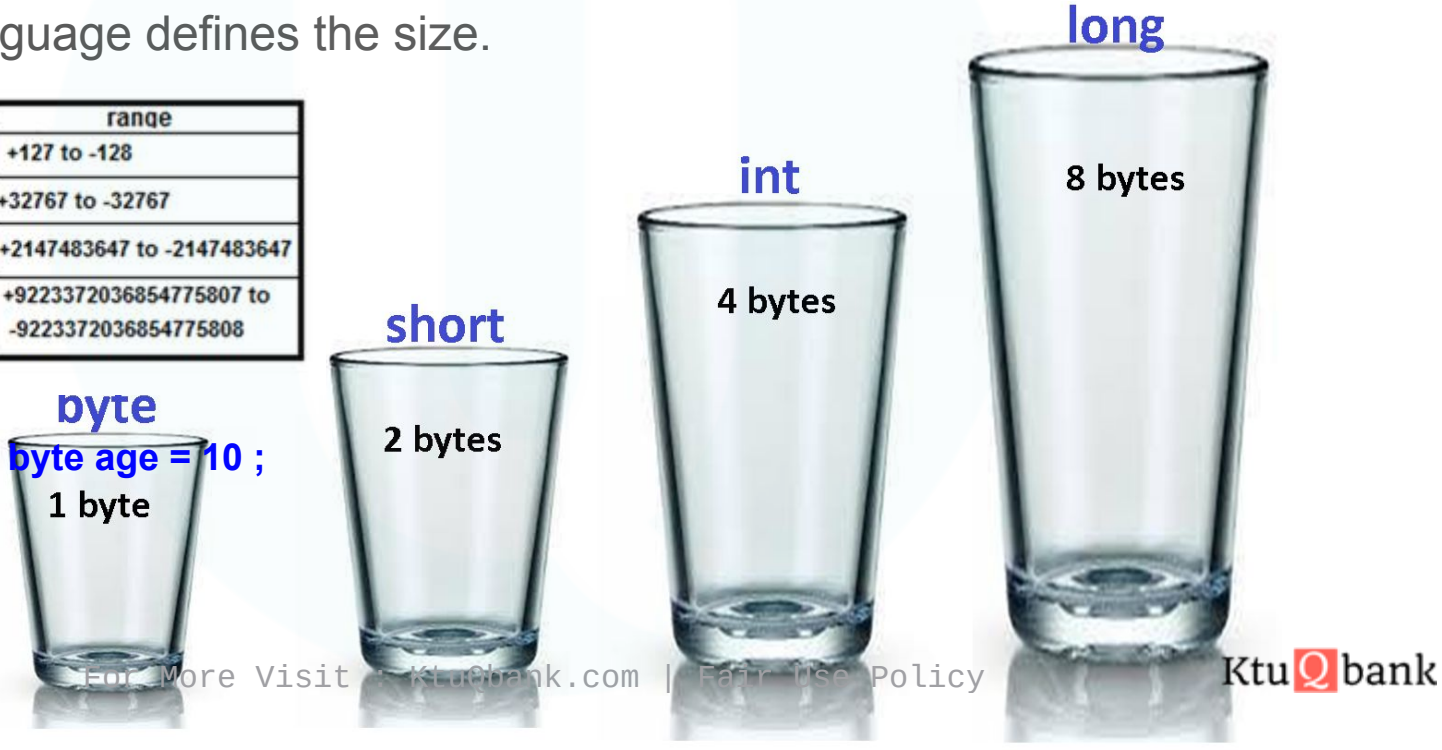
- A primitive type is predefined by the language and type name is a keyword.
- In Java there are 8 primitive data types: byte, short, int , long, char, float, double, and boolean
- Integers : byte, short, int , and long , which are for whole valued signed numbers.
- Floating point numbers : float and double , which represent numbers with fractional precision.
- Characters char , which represents symbols in a character set.
- Boolean boolean , represents true/false values.



Primitive data type - Integer

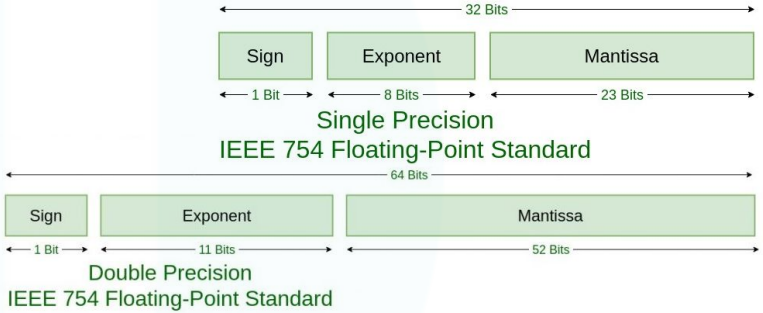
- The integer data types are used to store **whole numbers** like 2, 3 -1245 etc.
- There are 4 types to handle signed whole numbers
- In C/C++, size of int is dependant upon the platform.
- In Java, language defines the size.

sno	Data type	Size(in bytes)	range
1	byte	1	+127 to -128
2	short	2	+32767 to -32767
3	int	4	+2147483647 to -2147483647
4	long	8	+9223372036854775807 to -9223372036854775808



Primitive data type - Floating point

- Floating-point numbers: This group includes **float** and **double**, which represent numbers with fractional precision
- Java implements the standard (IEEE–754) set of floating-point types
- **float** - single-precision 32-bit
- **double** - precision 64-bit format



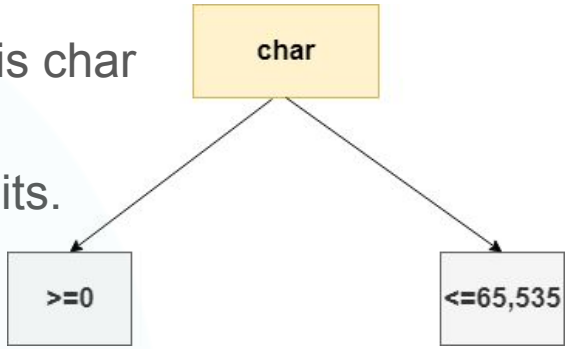

```
float f = 10.111111111;
```

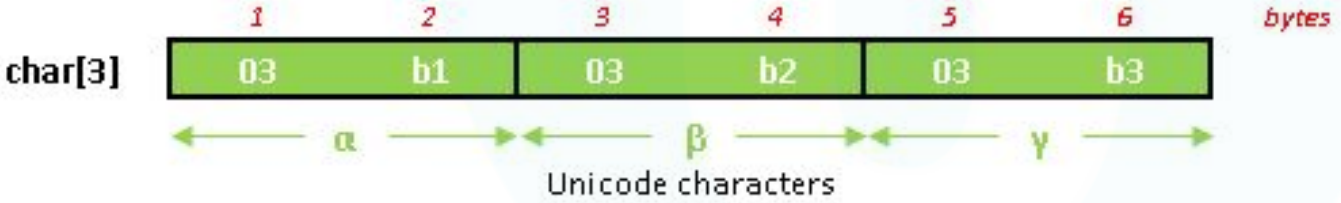
```
float f = 10.111111111f;
System.out.println(f);
output : - 10.111111 (6)
```

```
double d = 10.111111111;
System.out.println(d);
output : - 10.111111111(15)
```

Primitive data type - Character

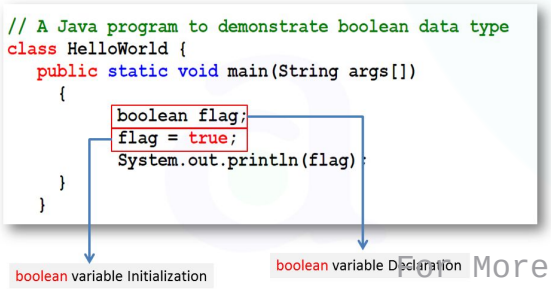
- In Java, the char data type used to store characters is char
- Java uses Unicode to represent characters
- At the time of Java's creation, Unicode required 16 bits. Thus, in Java char is a 16 bit type.
- Example: **char letter = 'A';**





Primitive data type - Boolean

- boolean: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions.
- This data type represents one bit of information, but its "size" isn't something that's precisely defined
- Size of boolean data type is virtual machine-dependent. Usually 1 byte.
- Values of type boolean are not converted implicitly or explicitly (with casts) to any other type.
- This is the type returned by all relational operators, as in the case of a < b .
- Example: boolean b;



Java Wrapper Classes

- Primitive type variables and not objects.
- Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.
- Primitive wrapper classes are used to create an Object that needs to represent primitive types in java

Datatypes	Size	Default value	Wrapper class
byte	8 bits	0	Byte
boolean	1 bit	False	Boolean
int	32 bits	0	Integer
char	16 bits	\u0000	Character
float	32 bits	0.0f	Float
double	64 bits	0.0d	Double
long	64 bits	0L	Long
short	16 bits	0	Short

Literals

- Primitive types are special data types built into the language; they are not objects created from a class.
- A literal is the source code representation of a fixed value, that do not change during the execution; literals are represented directly in your code without requiring computation.
- As shown below, it's possible to assign a literal to a variable of a primitive type
  - boolean result = true;
  - char capitalC = 'C'
  - byte b = 100;
  - short s = 10000;
  - int i = 10 ;

Integer Literal	100	0	27530	18
Floating Point Literal	3.14	-0.536	26783.087	0.9
String Literal	"India"	"838365"	"Good"	"cat"
Character Literal	'A'	'a'	'#'	')'
Boolean Literal	true	false		

Integer literals

- Integer literal are used to represent a **decimal** , **binary** , or **hexadecimal** value.
- Integer literals are used to initialize variables of integer data types **byte, short, int** and **long.**
- Integer literal that ends with **l** or **L** is of type **long** .
- In Java, a binary literals starts with **0b** , and hexadecimal literals starts with
- **0x** and rest every integer literal is a decimal literals
- int decInt = 18; // 18 in decimal notation
- int binInt = 0b10010; // 18 in binary notation
- int hexInt = 0x12; // 18 in hexadecimal notation
- long longValue = 123456789L ;

Floating point literals

- A floating point literal is of type float if it ends with the letter F or f
- Otherwise its type is double and it can optionally end with the letter D or d
- The floating point types (float and double) can also be expressed using E or e (for scientific notation
- **double myDouble = 123.4 ;**
- **double myDoubleScientific = 1.234e2 ;// same value as d1, but in scientific notation**
- **float myFloat = 123.4f ;**

Character and String literals

- Literals of types char and String may contain any Unicode (UTF 16) characters.
- String literals are represented as a sequence of characters surrounded by double quotes
- character literal is represented as a single character surrounded by single quotes.
- Java also allows use of **escape sequences** in string and character literals. [\b (backspace), \t (tab), \n (line feed), \f (form feed), \r (carriage return), \” (double quote), \’ (single quote), and \\(backslash]
- char myChar = 'A '; // Character Literal
- char newLine = n ';';// Character Literal Escape sequence
- String myString = "Java Tutorial "; // String Literal

Underscore characters in numeric literals

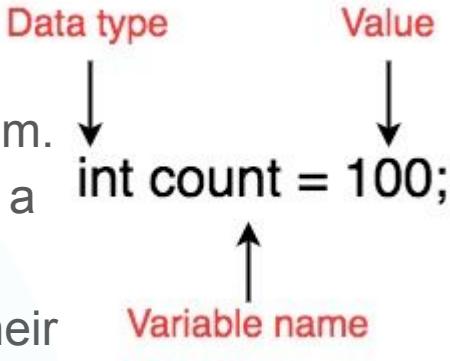
- In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal.
- This feature enables you, for example. to separate groups of digits in numeric literals, which can improve the readability of your code
- The following example shows other ways you can use the underscore in numeric literals:
  - long creditCardNumber = 1234_5678_9012_8906L ;
  - long socialSecurityNumber = 999_99_990 ;
  - float pi = 3.14_15F;
  - long hexBytes = 0xFF_EC_DE_AB ;
  - long hexWords = 0xCAFE_AABB ;
  - long maxLong = 0x7fff_ffff_ffff_ffffL ;
  - byte nybbles = 0b0010_0101 ;
  - long bytes = 0b11010010_01101001_10010100_10010010;

Underscore characters in numeric literals

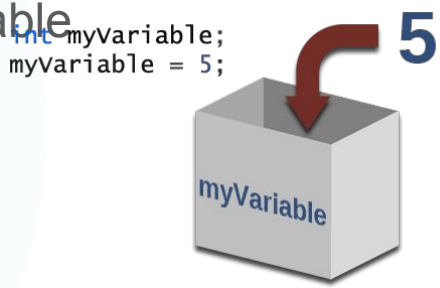
- You can place underscores only between digits; you cannot place underscores in the following places
- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an F or L suffix
- In positions where a string of digits is expected

```
// Invalid: cannot put underscores adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;
// Invalid: cannot put underscores in the 0x radix prefix
int x4 = 0_x52;
// Invalid: cannot put underscores at the beginning of a number
int x5 = 0x_52;
// OK (hexadecimal literal)
int x6 = 0x5_2;
// OK (decimal literal)
int x3 = 5_______2;
```

Variables



- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier , a type , and an optional initializer .
- In addition, all variables have a **scope** , which defines their visibility, and a **lifetime** which indicates how long the variable stays alive in the memory
- Here are several examples of variable declarations of various types. Note that some include an initialization
  - int a, b, c; // declares three ints , a, b, and c
  - int d = 3, e, f = 5; // declares three more ints , initializing d and f.
  - byte z = 22; // initializes
  - String Name = “Samia” ;



Variable naming rule

- Variable names are case sensitive.
- A variable's name can be any legal **identifier**
- The only allowed characters for identifiers are all alphanumeric characters (characters([A Z a z 0 9]), ‘$‘(dollar sign) and ‘_‘
- For example “geek@” is not a valid java identifier as it contain ‘@’ special character, geek$ is valid.
- Identifiers should not start with digits([0 9])9]).
- Reserved Words can’t be used as an identifier.
- For example “ int while = 20;” is an invalid statement as while is a reserved word.

Variable naming convention

- If the name you choose, consists of only one word, spell that word in all lowercase letters.
- If it consists of more than one word, capitalize the first letter of each subsequent word. This style is known as camel case.
- Examples
- gearRatio , currentGear
- If your variable stores a constant value, such as static final int
- NUM_GEARS = 6 , the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character.

Variable scope & life time

```
public class ToolBox
{
    public static double min ( double a, double b )
    {
        double m = 0;
        if ( a < b )
        {
            m = a;
        }
        else
        {
            m = b;
        }
        return(m);
    }
}
```

Lifetime of the parameter variables **a** *and* **b**

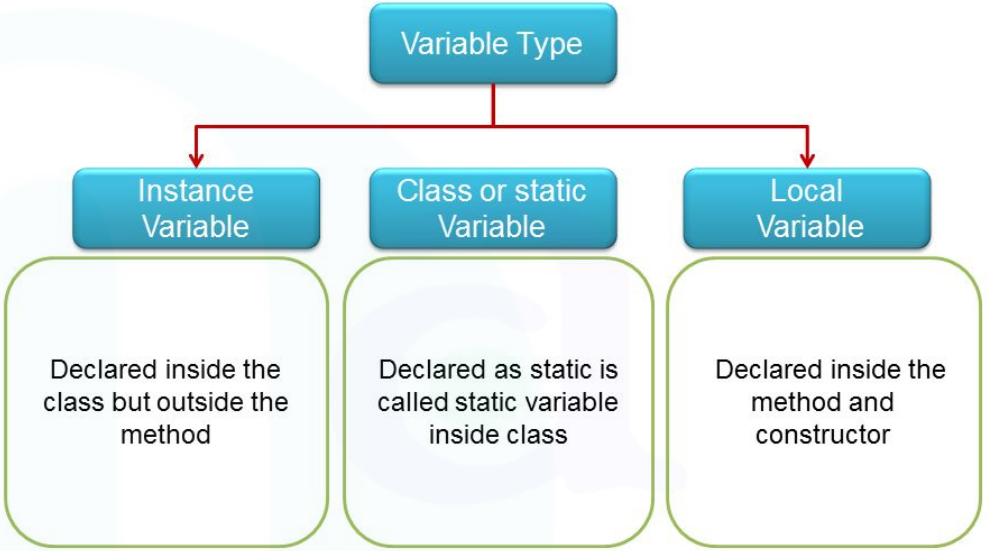
```
public class Scope2
{
   public static void main ( String[] args )
   {
      {
         double r;
         r = 3.14;
      }
      System.out.println( r );
   }
}
```

Scope of the variable **r**

use variable OUTSIDE its scope Error !!

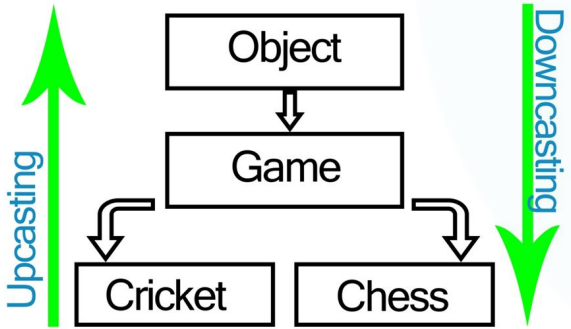
Types of Variables

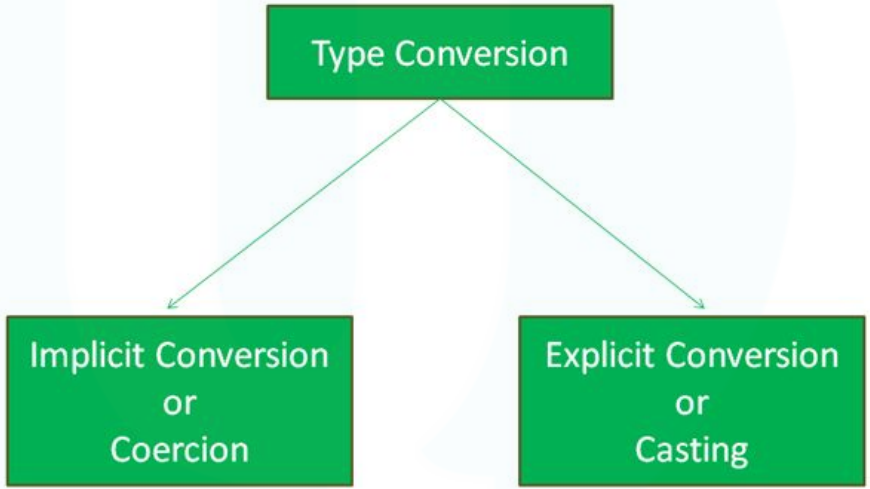
```
class A
{
    int data=50;   //instance variable
    static int m=100;  //static variable
    void method()
    {
        int n=90;  //local variable
    } //end of method
} //end of class
```



Type conversion and casting

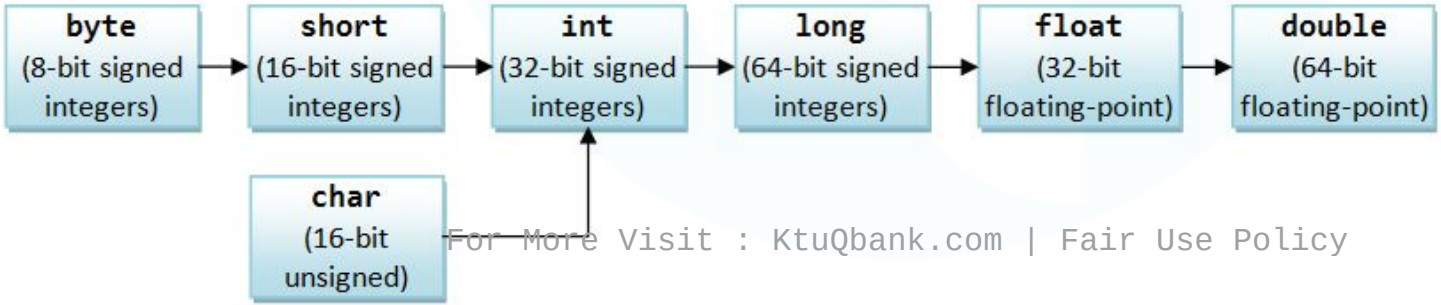
- When a data type is converted into another type is called type conversion
- When a variable is converted into a different type, the compiler basically treats the variable as of the new data type.
- Type of Type Conversion In Java
- Implicit Conversion
- Explicit Conversion



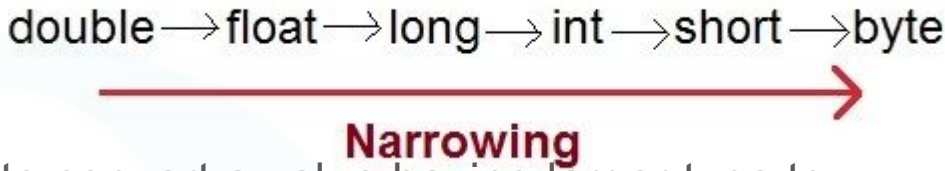


Implicit conversion.

- Implicit or automatic conversion compiler will automatically change one type of data into another without data loss.
- Implicit or automatic conversion can happen if both type are compatible and target type is larger than source type.
- If you assign an integer value to a floating point variable, the compiler will insert code to convert the int to a float.
- In implicit conversion a smaller type is converted into a larger thus it is also known as widening conversion.



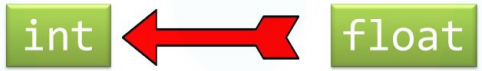
Explicit conversion.



- There may be situations when we want to convert a value having larger type to a smaller type.
- In this case casting needs to be performed explicitly.
- Explicit casting allows you to make this type conversion explicit, or to force it when it wouldn't normally happen.
- Syntax: (type) expression
- To perform type casting, put the desired type including modifiers inside parentheses to the left of the variable or constant you want to cast.

```
double d   = 10;
int i;
i = (int) d
```

Type Cast Operator



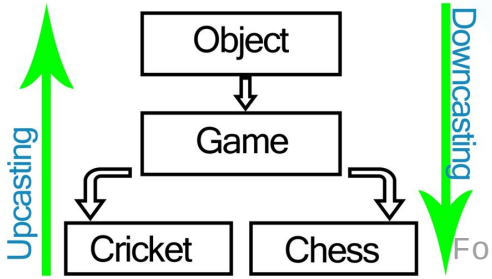

```
int number;
float fval= 32.33f;
number= (int)fval;
```

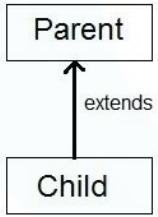
Type in which you want to convert

Variable name Which you want to convert

Upcasting & Downcasting.

- Just like the data types, the objects can also be type casted.
- In objects, there are only two types of objects (i.e.) parent object and child object.
- One type of object (i.e.) child or parent can be converted to another.
- There are two types of typecasting. They are:
- **Upcasting**: Upcasting is the typecasting of a child object to a parent object. Upcasting can be done implicitly.
- **Downcasting**: Similarly, downcasting means the typecasting of a parent object to a child object. Downcasting cannot be implicitly.





Parent p = new Child( );

Upcasting

Child c = new Parent( );

Compile time error

Child c = (Child) new Parent( );

Downcasting but throws ClassCastException at runtime.

References

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.
- Visual Paradigm - www.visual-paradigm.com
- GeeksforGeeks - www.geeksforgeeks.org
- Wikipedia - www.wikipedia.org
- Tutorialspoint - www.tutorialspoint.com
- Java Zone - https://dzone.com

CST 205 Object Oriented Programming using Java

CST 281 Object Oriented Programming
(As per KTU 2019 Syllabus)

Module 2 Lecture 4

Module 2 - Core Java Fundamentals

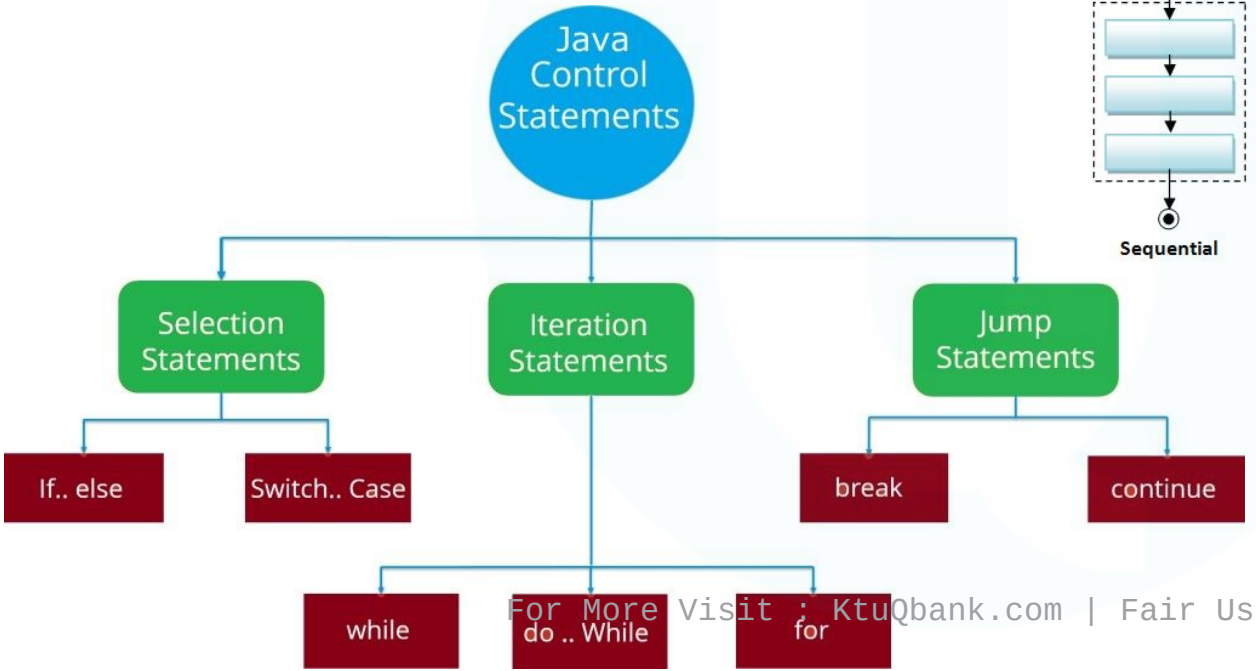
Primitive Data types - Integers, Floating Point Types, Characters, Boolean. Literals, Type Conversion and Casting, Variables, Arrays, Strings, Vector class. **Operators** - Arithmetic Operators, Bitwise Operators, Relational Operators, Boolean Logical Operators, Assignment Operator, Conditional (Ternary) Operator, Operator Precedence. **Control Statements** - Selection Statements, Iteration Statements and Jump Statements.

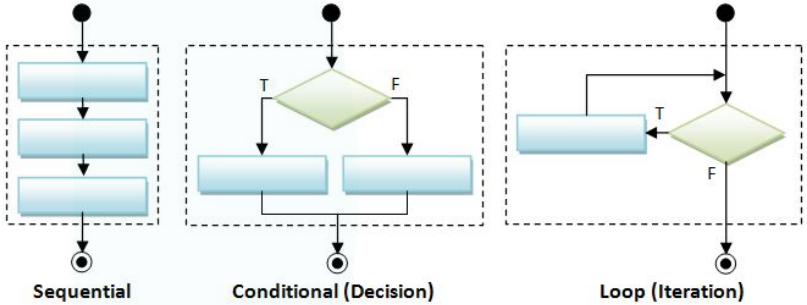
Object Oriented Programming in Java - Class Fundamentals, Declaring Objects, Object Reference, Introduction to Methods, Constructors, this Keyword, Method Overloading, Using Objects as Parameters, Returning Objects, Recursion, Access Control, Static Members, Final Variables, Inner Classes, Command Line Arguments, Variable Length Arguments.

Inheritance - Super Class, Sub Class, The Keyword super, protected Members, Calling Order of Constructors, Method Overriding, the Object class, Abstract Classes and Methods, using final with Inheritance.

Control Statements

- Default execution order of statements in a program is sequential, ie one after the other from top to bottom.
- Using control statement, this execution sequence of the program can be altered.
- A control statement in java is a statement that determines whether the other statements will be executed or not.





Selection Statements

- In java, the selection statements are also known as decision making statements or branching statements or conditional control statements.
- The selection statements are used to select a part of the program to be executed based on a condition.
- Java provides the following selection statements.

1. if statement
2. if-else statement
3. nested if statement
4. if-else if statement (if else if ladder)
5. switch statement

Selection if statement

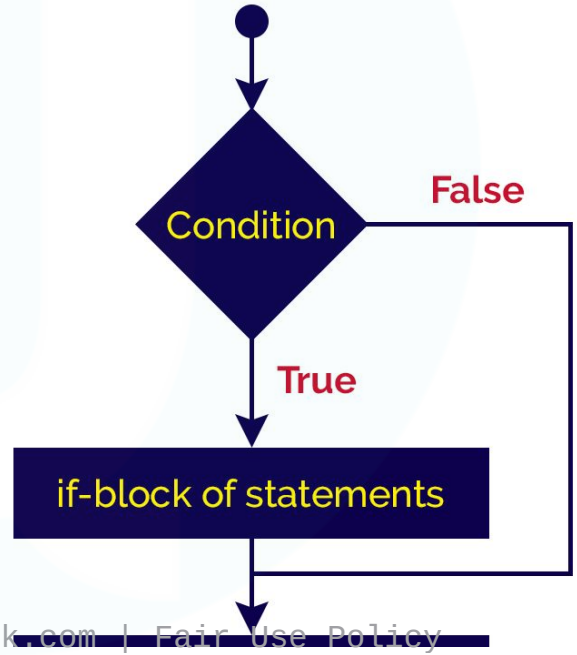
- In java, we use the if statement to test a condition and decide the execution of a block of statements based on that condition result.

Syntax

```
if(condition){
        if-block of statements;
        ...
}
statement after if-block;
...
```

```
if((num % 5) == 0) {
    System.out.println("We are inside the if-block!");
    System.out.println("Given number is divisible by 5!!");
}
```

Flow of execution



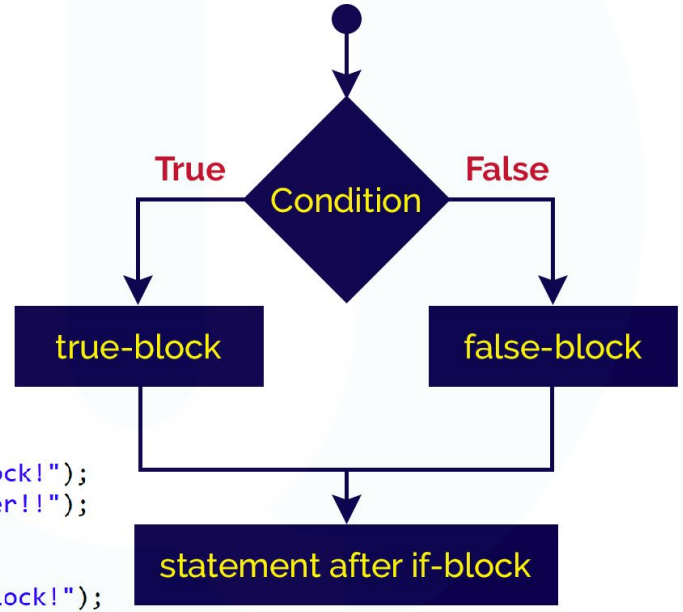
Selection: if-else statement

- if-else statement to test a condition and pick the execution of a block of statements out of two blocks based on that condition result

Syntax

```
if(condition){
        true-block of statements;
        ...
}
else{
        false-block of statements;
        ...
}
statement after if-block;
```

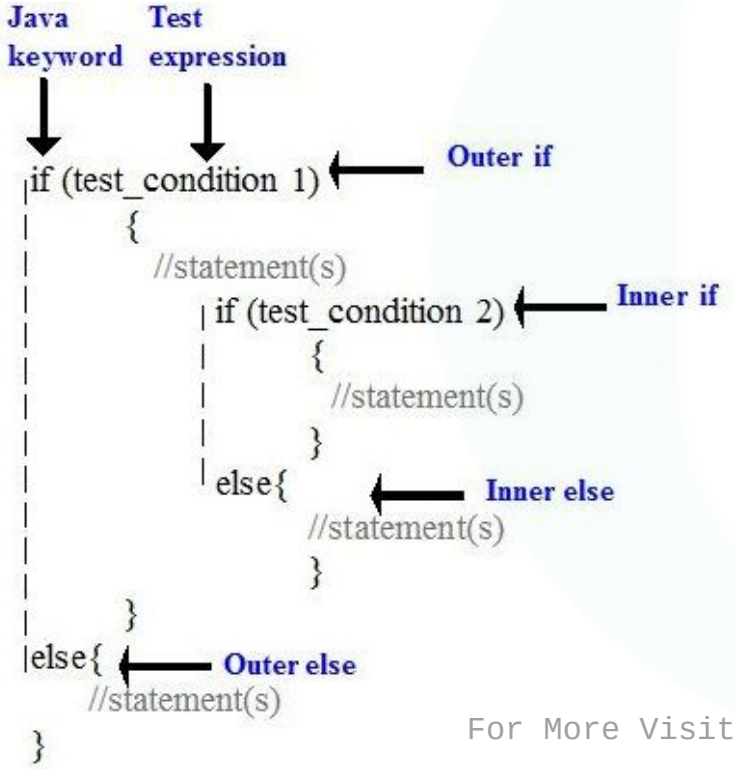
Flow of execution

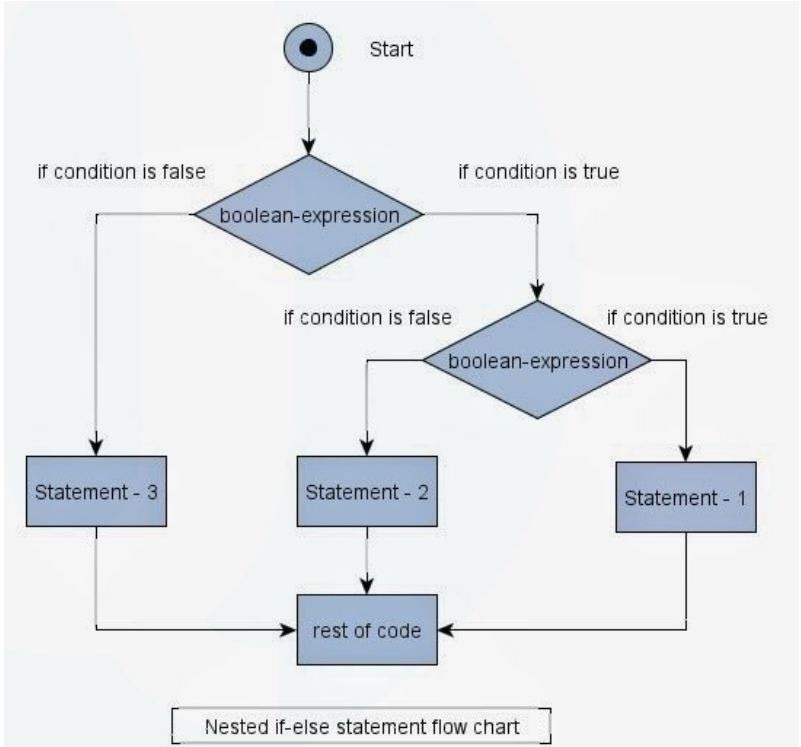



```
if((num % 2) == 0) {
    System.out.println("We are inside the true-block!");
    System.out.println("Given number is EVEN number!!");
}
else {
    System.out.println("We are inside the false-block!");
    System.out.println("Given number is ODD number!!");
}
```

Selection: nested if statement

- Writing an if statement inside another if-statement is called nested if statement.

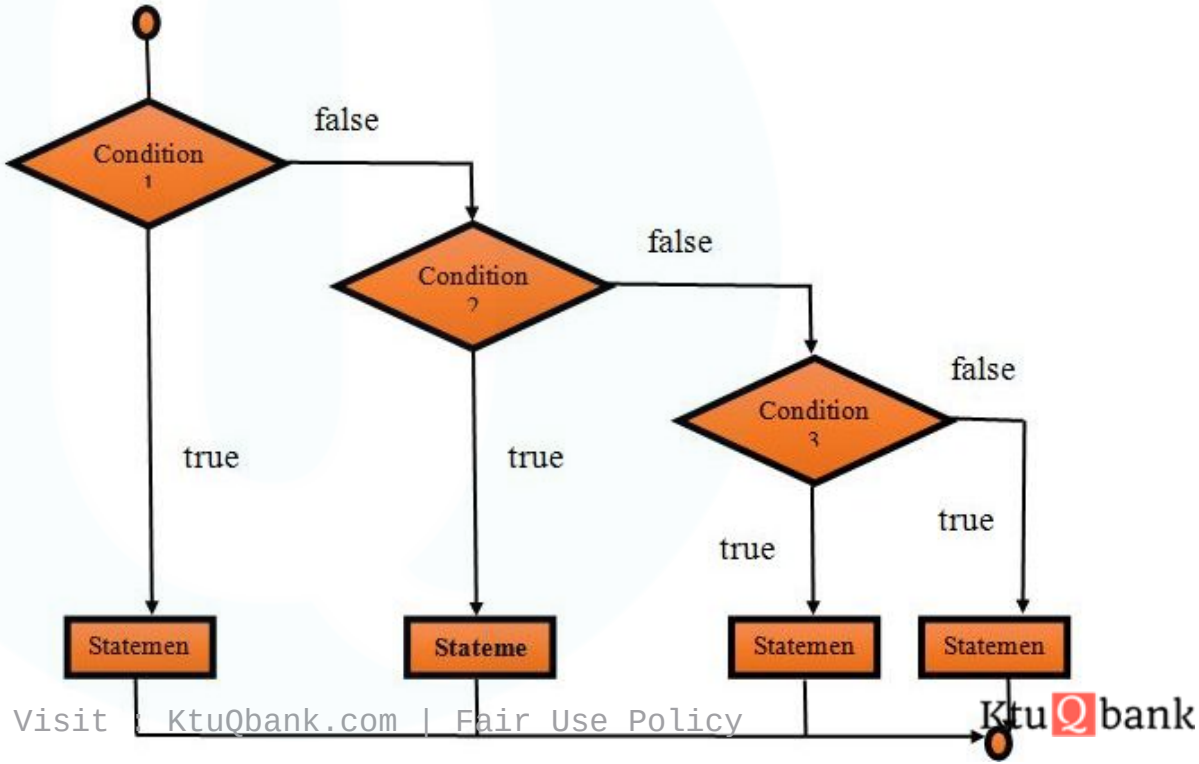




Selection: if else if (if else if ladder)

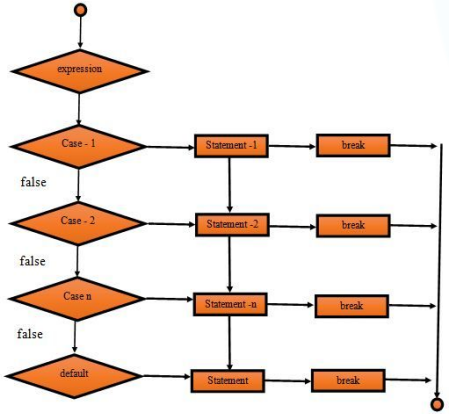
- The if-else-if ladder statement executes one condition from multiple statement block.

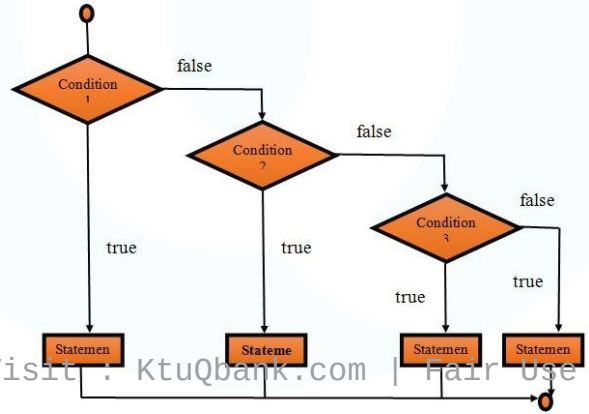
```
if(condition1)
{
//code for if condition1 is true
}
else if(condition2)
{
//code for if condition2 is true
}
else if(condition3)
{
//code for if condition3 is true
}
...
else
{
//code for all the false conditions
}
```



Selection: switch statement

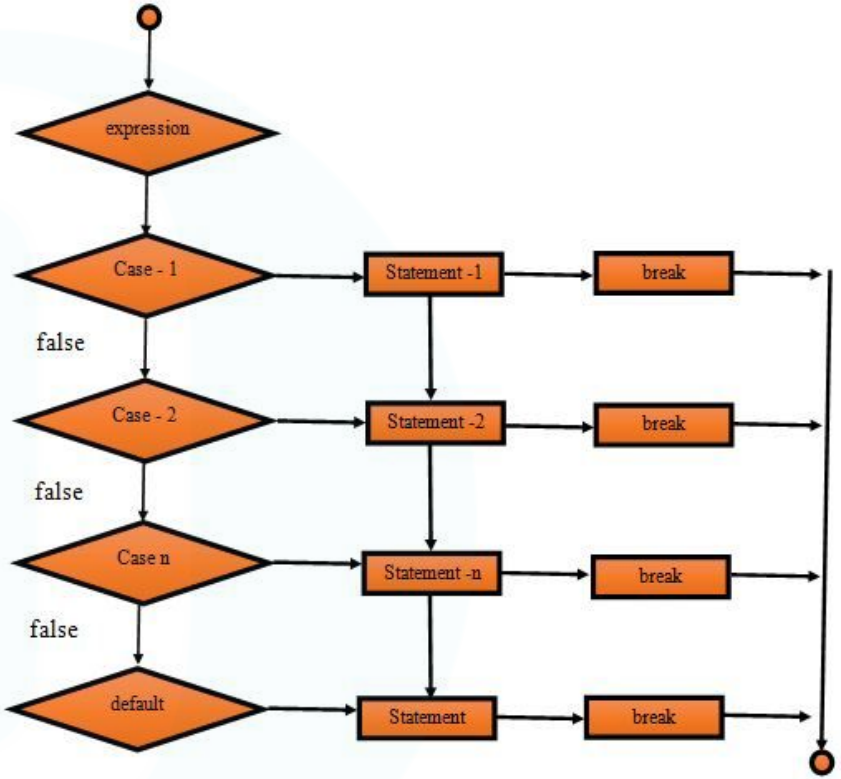
- Switch statement is used for executing one statement from multiple conditions.
- It is similar to an if-else-if ladder.
- In a switch statement,the expression can be of **byte, short, char** and **int** data types.
- From JDK7 **enum**, **String** class and the **Wrapper** classes can also be used.
- Following are some of the rules while using the switch statement:
  - There can be one or N numbers of cases.
  - The values in the case must be unique.
  - Each statement of the case can have a break statement. It is optional.





Selection: switch statement

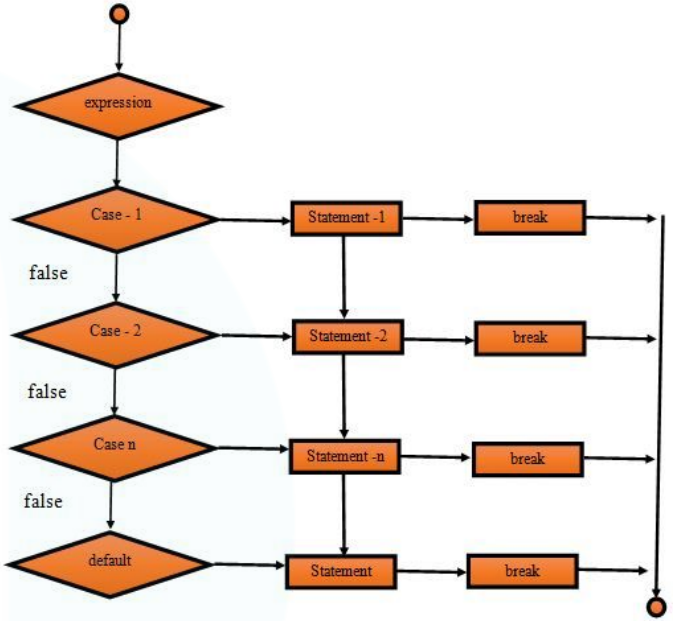
```
switch(expression)
{
case value1:
            //code for execution;
            break;  //optional
case value2:
 // code for execution
 break;  //optional
......
......
......
......
Case value n:
// code for execution
 break;  //optional

default:
 code for execution when none of the case is true;
}
```



Selection: switch statement

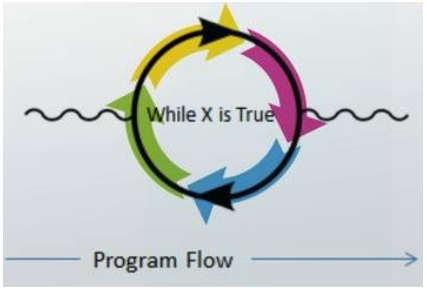
```
public class SwitchExample {

    public static void main(String[] args) {
        int n = 1;
        switch (n) {
            case 1:
                System.out.println("Khanh");
                break;
            case 2:
                System.out.println("Huong Dan Java");
                break;
            case 3:
                System.out.println("Lap Trinh Java");
                break;
            default:
                break;
        }
    }
}
```

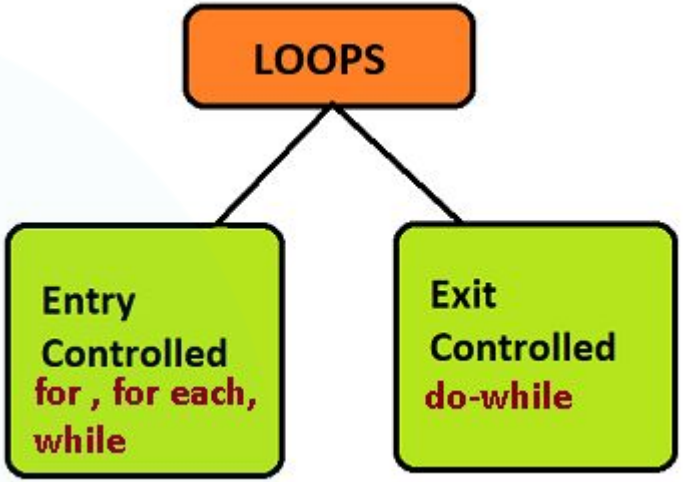
```
Khanh
```

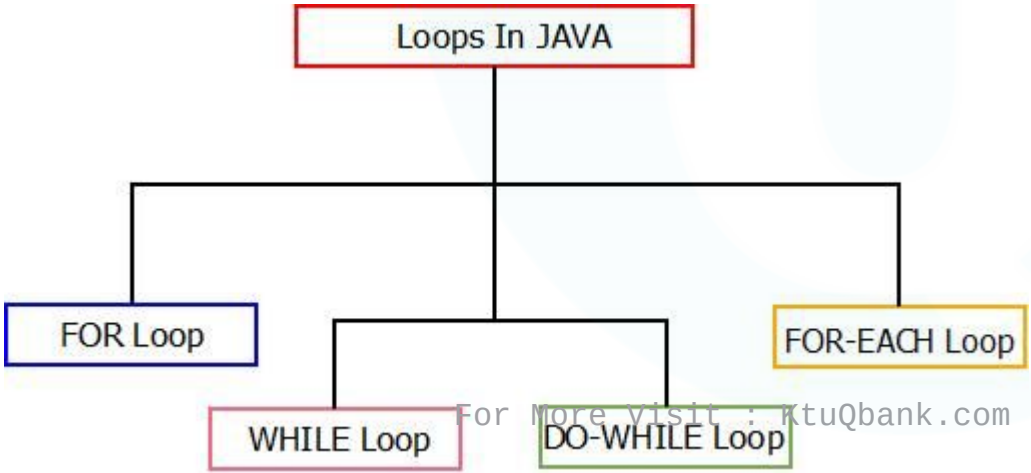


Iteration Statements (Loops)

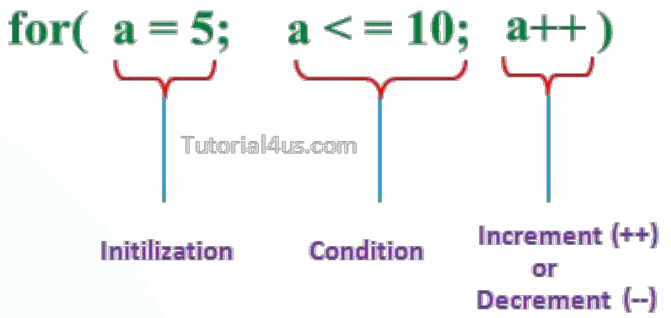
- A loop statement allows us to execute a statement or group of statements multiple times



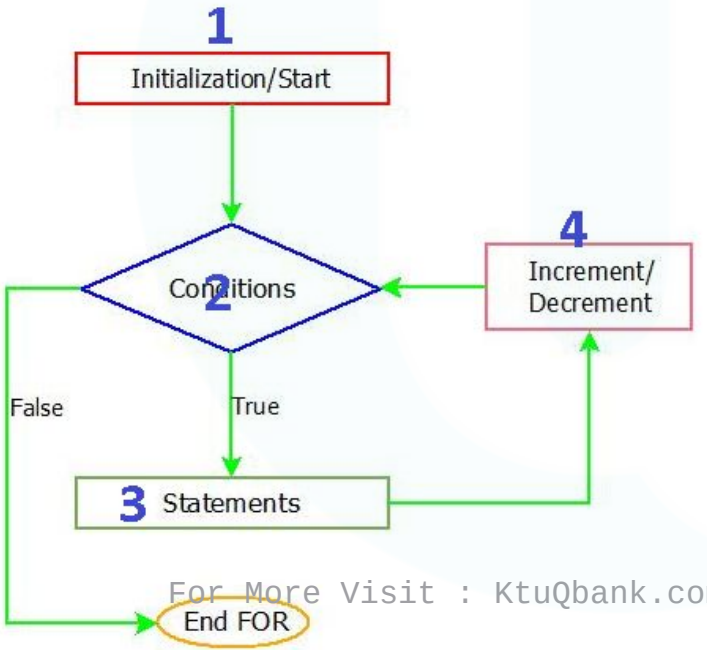




Iteration:for loop

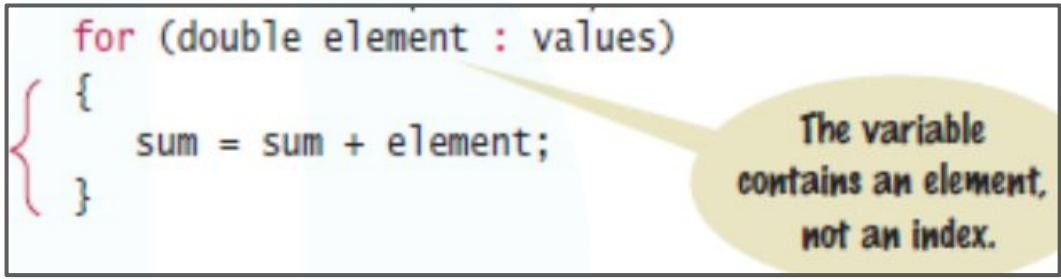



```
public static void main(String args[]) {
                1      2      4
    for (int i = 0; i < 10; i++) {

     3  System.out.println("The value of i is: " + i);

    }
}
```



Enhanced for loop (for each)

- As of Java 5, the enhanced for loop was introduced.
- This is mainly used to traverse collection of elements including arrays
- Syntax of enhanced for loop.
  - **for(declaration : expression)**
  - **{**
  - **// Statements**
  - **}**



- **Declaration** − The newly declared block variable, is of a type compatible with the elements of the array you are accessing.
- This variable will be available only within the for block and its value would be the same as the **current array element**.
- **Expression** − This evaluates to the array you need to loop through.
- The expression can be an array variable or method call that returns an array.

Enhanced for loop (for-each)

```
public class Test {

   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ) {
         System.out.print( x );
         System.out.print(",");
      }
      System.out.print("\n");
      String [] names = {"James", "Larry", "Tom", "Lacy"};

      for( String name : names ) {
         System.out.print( name );
         System.out.print(",");
      }
   }
}
```

Output

```
10, 20, 30, 40, 50,
James, Larry, Tom, Lacy,
```

Notes on for-each

- For-each loops are not appropriate when you want to modify the array.
- For-each loops do not keep track of index. So we can not obtain array index using For-Each loop.
- For-each only iterates forward over the array in single steps (ie, i+2 is not possible)
- For-each cannot process two decision making statements at once

```
// cannot be easily converted to a for-each loop
for (int i=0; i<numbers.length; i++)
{
   if (numbers[i] == arr[i]) { ...
   }
}
```

```
for (type var : array)
{
    statements using var;
}
```

is equivalent to:

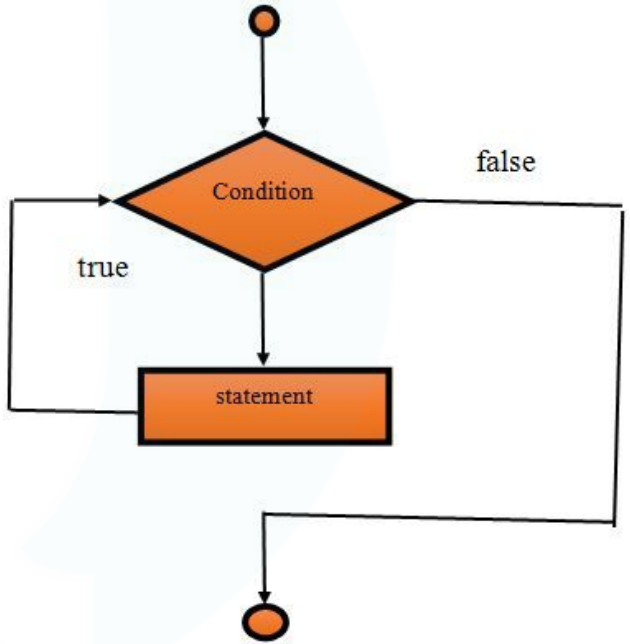
```
for (int i=0; i<arr.length; i++)
{
    type var = arr[i];
    statements using var;
}
```

hidden

while loop

- Entry controlled loop to repeat the loop body.
- When the number of iteration is not fixed then while loop is used

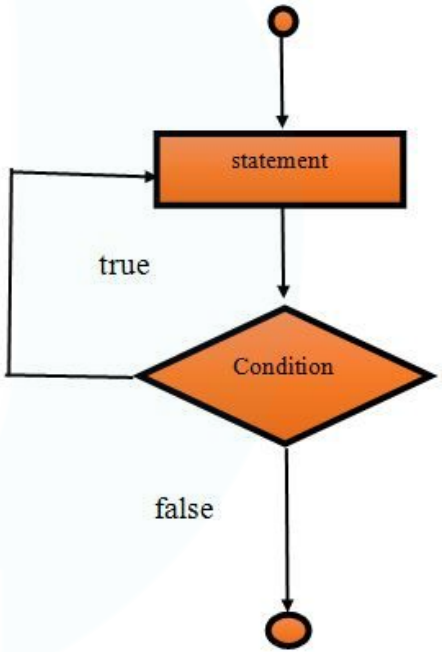
```
public class WhileDemo1
{
    public static void main(String[] args)
    {
        inti=1;
        while(i<=10)
        {
            System.out.println(i);
            i++;
        }
    }
}
```



do while loop

- Exit controlled loop to repeat the loop body.
- Irrespective of the loop condition, loop body will be executed at least once.

```
public class DoWhileDemo1
{
public static void main(String[] args)
{
inti=1;
do
{
System.out.println(i);
i++;
}while(i<=10);
        }
}
```



Nested loops

- A nested loop is a (inner) loop that appears in the loop body of another (outer) loop.
- The inner or outer loop can be any type: while, do while, or for.
- For example, the inner loop can be a while loop while an outer loop can be a for loop. Of course, they can be the same kind of loops too.

Outer loop / Inner loop

```
while (count++ < 50)
{
        // do work in outer loop
        // ...

        for (int i = 0; i < 25; i++)
        {
                // do work in inner lo
                // ...
        }
}
```

Outer Loop / Inner Loop

```
for (int row = 1; row <= 3; row++)
{
    for (int col = 1; col <= 5; col++)
    {
        System.out.print("*");
    }
     System.out.println();
}
```

Labeled statements

- A label is a name attached to a statement. [**labele_name :** statement…]
- A **goto** statement in C/C++ programming provides an unconditional jump from the 'goto' to a **labeled** statement in the same function
- Java does not have goto statement, but java supports label.
- The only place where a label is useful in Java is **right before nested loop**.
- You can attach a label name just before the beginning of a loop.
- Label name can be used with break/continue statement for unconditional jump.

```
#include<stdio.h>
#include <conio.h>
int main()
{
printf ("Hello World\n");
goto Label;
printf("How are you?");
printf("Are you Okey?");
Label:
    printf("Hope you are fine");
getch();
}
```

Jumped

Skipped

```
outer: while(condition)
       {
         if(condition)
           break outer;

         statement2;
       }
statement3;
```

while loop is labelled as "outer" and hence this statement "break outer" breaks the control out of the loop named "outer", without executing statement2.

Jump statements

- Loop control statements change execution from its normal sequence.
- When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

break statement

- Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
- Has two forms: labeled and unlabeled.

continue statement

- Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
- Has two forms: labeled and unlabeled.

return statement

break;

```
while (testExpression) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

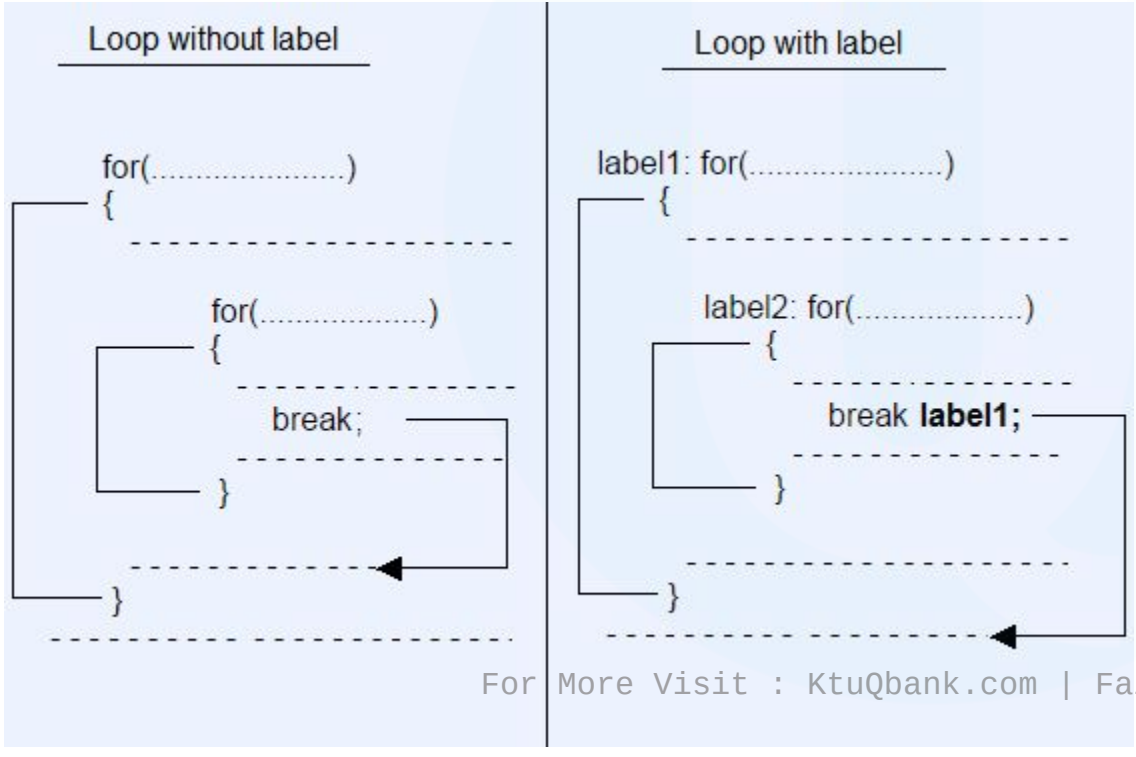
```
do {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

```
while (testExpression) {
    // codes
    while (testExpression) {
        // codes
        if (condition to break) {
            break;
        }
        // codes
    }
    // codes
}
```

Labeled break;

- According to nested loop, if we put break statement in inner loop, compiler will jump out from inner loop and continue the outer loop again.
- To jump out from the outer loop, we can use labeled break.



continue;

```
while (testExpression) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

```
do {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
} while (testExpression);
```

```
while(testExpresison) {
    // codes
    while (testExpression) {
        // codes
        if (condition for continue) {
            continue;
        }
        // codes
    }
    // codes
}
```

```
for (init; testExpression; update) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

Labeled continue;

- In the case of nested loops, the continue skips the current iteration of the innermost loop.
- To skip the current iteration of the outer loop, labeled continue can be used.

```
label:
while (testExpression) {
    // codes
    while (testExpression) {
        // codes
        if (condition for continue) {
            continue label;
        }
        // codes
    }
    // codes
}
```

```
while(testExpresison) {
    // codes
    while (testExpression) {
        // codes
        if (condition for continue) {
            continue;
        }
        // codes
    }
    // codes
}
```

return statement

- The **return** statement exits from the current method, and control flow returns to where the method was invoked.
- The return statement has two forms: one that returns a value, and one that doesn't.
- To return a value, simply put the value (or an expression that calculates the value) after the return keyword.
- **return ++count;**
- The data type of the returned value must match the type of the method's declared return value.
- When a method is declared void, use the form of return that doesn't return a value.
- **return;**

```
public class ReturnValue
{
    public int addition(int a,int b)
    {
        int sum = a+b;
        return sum;
    }

    public static void main(String args[])
    {
        ReturnValue r = new ReturnValue();
        int answer;

        answer = r.addition(10,20);
        System.out.println("Addition = "+answer);
    }
}
```

References

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.
- Visual Paradigm - www.visual-paradigm.com
- GeeksforGeeks - www.geeksforgeeks.org
- Wikipedia - www.wikipedia.org
- Tutorialspoint - www.tutorialspoint.com
- Java Zone - https://dzone.com

CST 205 Object Oriented Programming using Java

CST 281 Object Oriented Programming
(As per KTU 2019 Syllabus)

Module 2 Lecture 3

Module 2 - Core Java Fundamentals

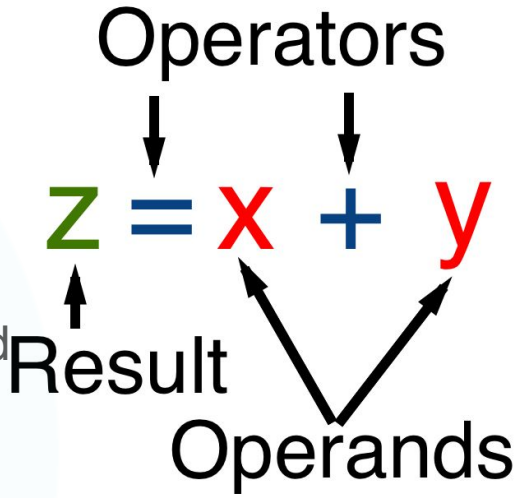
Primitive Data types - Integers, Floating Point Types, Characters, Boolean. Literals, Type Conversion and Casting, Variables, Arrays, Strings, Vector class. **Operators** - Arithmetic Operators, Bitwise Operators, Relational Operators, Boolean Logical Operators, Assignment Operator, Conditional (Ternary) Operator, Operator Precedence. **Control Statements** - Selection Statements, Iteration Statements and Jump Statements.

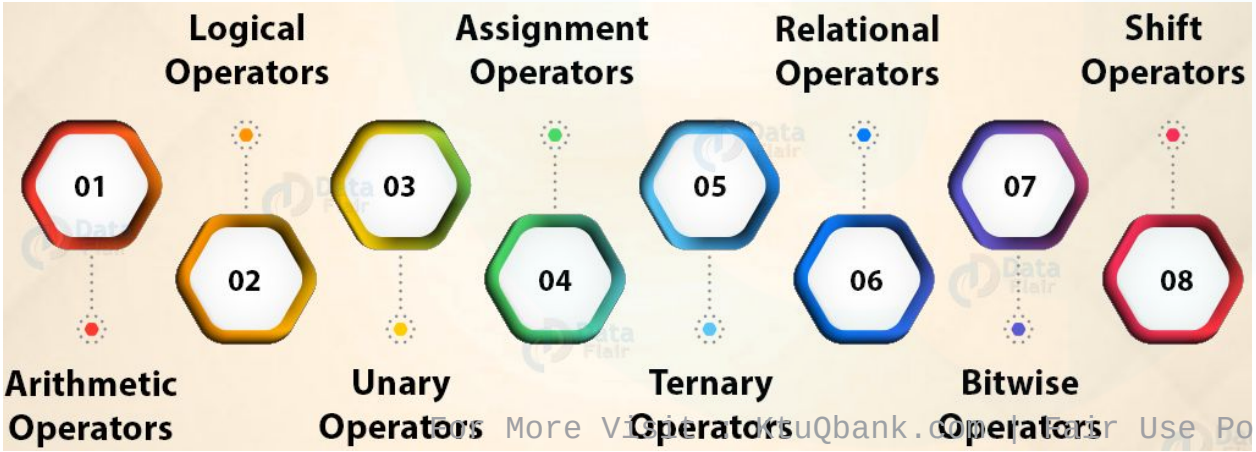
Object Oriented Programming in Java - Class Fundamentals, Declaring Objects, Object Reference, Introduction to Methods, Constructors, this Keyword, Method Overloading, Using Objects as Parameters, Returning Objects, Recursion, Access Control, Static Members, Final Variables, Inner Classes, Command Line Arguments, Variable Length Arguments.

Inheritance - Super Class, Sub Class, The Keyword super, protected Members, Calling Order of Constructors, Method Overriding, the Object class, Abstract Classes and Methods, using final with Inheritance.

Operators in Java

- Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.
- An operator is a token that tells the computer to perform certain mathematical or logical operations on variables and literals.





Arithmetic operators

Operator	Meaning	Example	Result
+	Addition	10 + 2	12
-	Subtraction	10 – 2	8
*	Multiplication	10 * 2	20
/	Division	10 / 2	5
%	Modulus (remainder)	10 % 2	0
++	Increment	a++ (consider a = 10)	11
--	Decrement	a-- (consider a = 10)	9
+=	Addition Assignment	a += 10 (consider a = 10)	20
-=	Subtraction assignment	a -= 10 (consider a = 10)	0
*=	Multiplication assignment	a *= 10 (consider a = 10)	100
/=	Division assignment	a /= 10 (consider a = 10)	1
%=	Modulus assignment	a %= 10 (consider a = 10)	0

Arithmetic operators - Compound operators

- a+=10 is same as that of a = a + 10
- += is an example for Compound Assignment Operators, also known as Shorthand Assignment Operators.
- It's called shorthand because it provides a short way to assign an expression to a variable.
- This operator can be used to connect Arithmetic operator with an Assignment operator

Modulus operator (%)

A = 5, B =2

C = A%B

C = ?

```
import java.util.*;

public class Demo{
      public static void main(String[] args){

          double x = 4.3;
          double y = 1.1;

          System.out.println(x%y);


      }
}
```

```
0.9999999999999996
```

++ / --

Initial x	Expression	Final y	Final x
3	y=x++	3	4
3	y=++x	4	4
3	y=x--	3	2
3	y=--x	2	2

X++ is Use –Then - Change

++x is Change – Then - Use

- A++ is same as A = A + 1
- A-- is same as A = A - 1
- ++A is pre increment, change first then use the value when there is an assignment.
- A++ post increment
- Use the value then change (when there is an assignment)

Assignment operator

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

- Assignment operators are used to assign values to variables.
- Eg: int A = C + 10 ;
- When the target variable and one of the operand are same then we can use shorthand / compound operators. (Not applicable for ternary operators)

Unary Operators

- Java unary operators are the types that need only one operand to perform any operation.
- It consists of various arithmetic, logical and other operators that operate on a single operand

+	Unary plus; indicates positive value
-	Unary minus; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

```
int x = 5;
int a, b;
a = x++;        // a = 5 x = 6
x--;            // x = 5
b = ++x;        // b = 6 x = 6
++x;            // x = 7
boolean bool = true;
// true
System.out.println(bool);
// false
System.out.println(!bool);
```

Java Comparison/Relational Operators

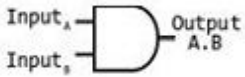
- Comparison operators are used to compare two values
- Produces Boolean result

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Java Boolean Logical Operators

- Logical operators are used to determine the logic between variables or values.
- Produces boolean result.
- In Java (&&) AND and (||) OR supports **short circuiting**

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)



Truth table: AND

A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

Truth table: OR

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Truth table: NOT

A	~A
0	1
1	0

Java Bitwise Operators

- Bitwise operators are used to perform binary logic with the bits of an **integer** or **long** integer.

Operator	Description	Example	Same as	Result	Decimal
&	AND - Sets each bit to 1 if both bits are 1	5 & 1	0101 & 0001	0001	1
|	OR - Sets each bit to 1 if any of the two bits is 1	5 | 1	0101 | 0001	0101	5
~	NOT - Inverts all the bits	~ 5	~0101	1010	10
^	XOR - Sets each bit to 1 if only one of the two bits is 1	5 ^ 1	0101 ^ 0001	0100	4
<<	Zero-fill left shift - Shift left by pushing zeroes in from the right and letting the leftmost bits fall off	9 << 1	1001 << 1	0010	2
>>	Signed right shift - Shift right by pushing copies of the leftmost bit in from the left and letting the rightmost bits fall off	9 >> 1	1001 >> 1	1100	12
>>>	Zero-fill right shift - Shift right by pushing zeroes in from the left and letting the rightmost bits fall off	9 >>> 1	1001 >>> 1	0100	4

Note: The Bitwise examples above use 4-bit unsigned examples, but Java uses 32-bit signed integers and 64-bit signed long integers. Because of this, in Java, ~5 will not return 10. It will return -6. ~00000000000000000000000000000101 will return 11111111111111111111111111111010

In Java, 9 >> 1 will not return 12. It will return 4. 00000000000000000000000000001001 >> 1 will return 00000000000000000000000000000100

Java Shift Operators (Unsigned right shift)

- >>> Right shift with zero fill or unsigned right shift is a newly introduced operator in Java

```
int a =  3; // ...00000011 =  3
int b = -4; // ...11111100 = -4
```

>>> Right 0

```
a         00000000000000000000000000000011      3
a >>> 2   00000000000000000000000000000000      0

b         11111111111111111111111111111100     -4
b >>> 2   00111111111111111111111111111111     +big
```

<< Left

```
a         00000000000000000000000000000011      3
a << 2    00000000000000000000000000001100     12

b         11111111111111111111111111111100     -4
b << 2    11111111111111111111111111110000    -16
```

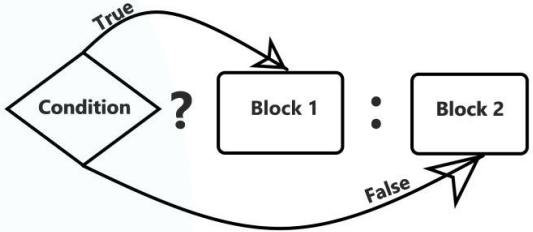
>> Right

```
a         00000000000000000000000000000011      3
a >> 2    00000000000000000000000000000000      0

b         11111111111111111111111111111100     -4
b >> 2    11111111111111111111111111111111     -1
```

Java Ternary Operato

```java
int age = 10;

String result = (age > 18) ? "Yes, you can vote!" : "No, you can't vote!";

System.out.println(result);
```

- As the name ternary suggests, it is the only operator in Java consisting of three operands.
- The ternary operator can be thought of as a simplified version of the if-else statement with a value to be returned.
- Syntax
  variable = (condition) ? block1 : block2 ;




```java
public class LargestOfThree {

    public static void main(String[] args) {

        int a = 55;
        int b = 95;
        int c = 75;

        int result = ( a > b ) ? (a > c ? a : c) : ( b > c ? b : c);

        System.out.println(result);

    }

}
```

Type Comparison Operator

- In Java, the **instanceof** keyword is a binary operator. Result will be boolean
- It is used to check whether an object is an instance of a particular class or not.
- The operator also checks whether an object is an instance of a class that implements an interface. Syntax: **object instanceof ClassName/InterfaceName;**

```
class Animal {
}

// Dog class is a subclass of Animal
class Dog extends Animal {
}

class Main {
    public static void main(String[] args){
        Dog d1 = new Dog();

        // checks if d1 is an object of Dog
        System.out.println("Is d1 an instance of Dog: "+ (d1 instanceof Dog));

        // checks if d1 is an object of Animal
        System.out.println("Is d1 an instance of Animal: "+ (d1 instanceof Animal));
    }
}
```

Output:

```
Is d1 is an instance of Dog: true
Is d1 an instance of Animal: true
```

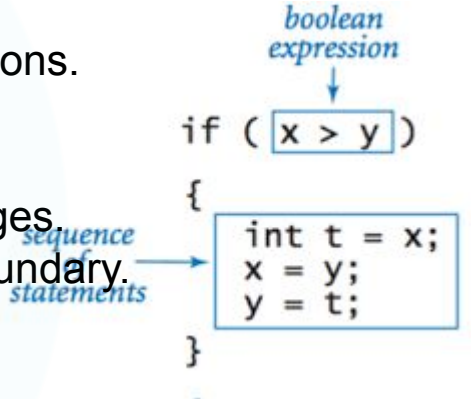
Expression, Statement, Block

Expressions

- An expression is a construct made up of variables, operators, and method invocations.
- Constructed according to the syntax of the language, that evaluates to a single value.
  - int result = 1 + 2 // result is now 3
- Compound expressions are created from various smaller expressions.
  - Eg: a = x + y / 100 // ambiguous

Statements

- Statements are roughly equivalent to sentences in natural languages.
- A statement forms a complete unit of execution. ; indicates the boundary.
- Expression statement
  - Bicycle myBike = new Bicycle();
- Declaration statements - A declaration statement declares a variable.
  - double aValue = 8933.234;
- Control flow statements regulate the order in which statements get executed.



Block

A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

Expression evaluation in Java

- When your Java program executes and encounters a line with an expression, the expression is *evaluated* (its value is computed).
  - The expression `3 * 4` is evaluated to obtain `12`.
  - `System.out.println(3 * 4)` prints `12`, not `3 * 4`.

Evaluate following expressions.

- A = 10 + 5 ;
- C = 10 * 2 - 3 ;

Evaluate 2*x-3*y ?

- (2x)-(3y) or 2(x-3y) which one is correct?

Evaluate A / B * C

- A / (B * C) or (A / B) * C Which one is correct?

To answer these questions satisfactorily one has to understand the priority or the order in which these operators are evaluated.

Operator Precedence

- **Operator precedence** determines the order in which the operators in an expression are evaluated.
- **Precedence order**. When two operators share an operand the operator with the higher precedence goes first.
  - For example, 1 + 2 * 3 is treated as 1 + (2 * 3), whereas 1 * 2 + 3 is treated as (1 * 2) + 3 since multiplication has a higher precedence than addition.
- **Associativity**. When an expression has two operators with the same precedence, the expression is evaluated according to its associativity.
  - For example x = y = z = 17 is treated as x = (y = (z = 17)), leaving all three variables with the value 17, since the **=** operator has **right-to-left** associativity (and an assignment statement evaluates to the value on the right hand side).
  - On the other hand, 72 / 2 / 3 is treated as (72 / 2) / 3 since the **/** operator has **left-to-right** associativity.
  - Some operators are not associative: for example, the expressions (x <= y <= z) and x++-- are **invalid**.

Operator Precedence

Order of evaluation of subexpressions.

- Associativity and precedence determine in which order Java applies operators to **expressions** but they do not determine in which order the **subexpressions** are evaluated.
- In Java, subexpressions are evaluated from **left to right** (when there is a choice).
- So, for example in the expression **A() + B() * C(D(), E())**, the subexpressions are evaluated in the order A(), B(), D(), E(), and C().
- It is considered poor style to write code that relies upon this behavior (and different programming languages may use different rules).

Short circuiting.

- When using the conditional and and or operators (&& and ||), Java does not evaluate the second operand unless it is necessary to resolve the result.
- This allows statements like if (s != null && s.length() < 10) to work reliably.
- Programmers rarely use the non short-circuiting versions (& and |) with boolean expressions.

Precedence table

Highest level to lowest

Level	Operator	Description	Associativity

16	`[]`
	`.`
	`()`
		access array element
		access object member
		parentheses
			left to right

15	`++` `--`	unary post-increment unary post-decrement	not associative

14	`++`
	`--`
	`+`
	`-`
	`!`
	`~`
		unary pre-increment
		unary pre-decrement
		unary plus
		unary minus
		unary logical NOT
		unary bitwise NOT
			right to left

13	`()` `new`	cast object creation	right to left
12	`* / %`	multiplicative	left to right
11	`+ -` `+`	additive string concatenation	left to right
10	`<< >>` `>>>`	shift	left to right

9	`< <=`
	`> >=`
	`instanceof`
		relational	not associative

8	`==` `!=`	equality	left to right
7	`&`	bitwise AND	left to right
6	`^`	bitwise XOR	left to right
5	`|`	bitwise OR	left to right
4	`&&`	logical AND	left to right
3	`||`	logical OR	left to right
2	`?:`	ternary	right to left

1	`=` `+=` `-=`
	`*=` `/=` `%=`
	`&=` `^=` `|=`
	`<<=` `>>=` `>>>=`
		assignment	right to left

Operator Precedence

```java
public class OperatorPrecedence {
    public static void main(String[] args) {
        int x = 5;
        int y = 10;
        int z = ++x * y--;
        int a = 7 * 3 + 24 / 3 - 5;
        int b = (7 * 3) + 24 / (3 - 5);
        System.out.println("1 + 2 = " + 1 + 2);
        System.out.println("1 + 2 = " + (1 + 2));
        System.out.print("a :" +a + " b:" + b + " z:" +z);
    }
}
```

```
D:\GECI\CST205-OOPJ\Java\basics>java Operato
1 + 2 = 12
1 + 2 = 3
a :24 b:9 z:60
D:\GECI\CST205-OOPJ\Java\basics>
```

References

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.
- Visual Paradigm - www.visual-paradigm.com
- GeeksforGeeks - www.geeksforgeeks.org
- Wikipedia - www.wikipedia.org
- Tutorialspoint - www.tutorialspoint.com
- Java Zone - https://dzone.com

ST. JOSEPH'S
COLLEGE OF ENGINEERING
AND TECHNOLOGY,
- PALAI -

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CST 205 - Object Oriented Programming Using Java

Prof. Sarju S

24 September 2020

Module 2: Core Java Fundamentals

Primitive Data types - Integers, Floating Point Types, Characters, Boolean. Literals, Type Conversion and Casting, Variables, **Arrays**, Strings, **Vector class**.

Operators - Arithmetic Operators, Bitwise Operators, Relational Operators, Boolean Logical Operators, Assignment Operator, Conditional (Ternary) Operator, Operator Precedence.

Control Statements - Selection Statements, Iteration Statements and Jump Statements.

Object Oriented Programming in Java - Class Fundamentals, Declaring Objects, Object Reference, Introduction to Methods, Constructors, this Keyword, Method Overloading, Using Objects as Parameters, Returning Objects, Recursion, Access Control, Static Members, Final Variables, Inner Classes, Command Line Arguments, Variable Length Arguments.

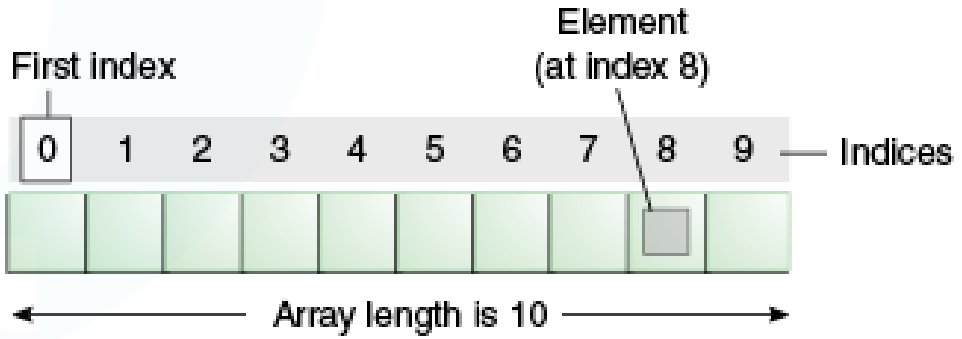
Inheritance - Super Class, Sub Class, The Keyword super, protected Members, Calling Order of Constructors, Method Overriding, the Object class, Abstract Classes and Methods, using final with Inheritance.

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Arrays in Java

Arrays in Java

- Arrays are objects which store multiple variables of the same type.
- It can hold primitive types as well as object references.
- Features of Array
  - Arrays are objects
  - They can even hold the reference variables of other objects
  - They are created during runtime
  - They are dynamic, created on the heap
  - The Array length is fixed



Source: https://docs.oracle.com/javase/tutorial/

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Arrays in Java

- To declare an array, define the variable type with square brackets:

```
// declares an array of integers
int[] anArray;
```

- You can also place the brackets after the array's name:

```
// this form is discouraged
float anArrayOfFloats[];
```

- However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Creating, Initializing, and Accessing an Array

- One way to create an array is with the new operator.

```
// declares an array of integers
int[] anArray;
// create an array of integers
anArray = new int[10];
```

- To assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // and so forth
```

- Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```


Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Creating, Initializing, and Accessing an Array

- Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

- Here the length of the array is determined by the number of values provided between braces and separated by commas.

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Array Length

- To find out how many elements an array has, use the length property:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);// Outputs 4
```


Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Loop Through an Array

- You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}
```

- There is also a "for-each“(Enhanced for) loop, which is used exclusively to loop through elements in arrays:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String car : cars) {
  System.out.println(car);
}
```

Multidimensional Arrays

- A multidimensional array is an array of arrays. Each element of a multidimensional array is an array itself. For example,

```
int[][] anArray = new int[3][4];
```

- To create a two-dimensional array, add each array within its own set of curly braces:

```
int[][] anArray = {
        {1, 2, 3},
        {4, 5, 6, 9},
};
// calculate the length of each row
System.out.println("Length of row 1: " + anArray[0].length);
System.out.println("Length of row 2: " + anArray[1].length);
```


Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Loop Through an 2D Array

- We can also use the for...each loop to access elements of the multidimensional array. For example,

```
public class MultidimensionalArrayDemo1 {

    public static void main(String[] args) {
        int[][] anArray = {
                {1, -2, 3},
                {-4, -5, 6, 9},
                {7},
            };

            for (int i = 0; i < anArray.length; ++i) {
                for(int j = 0; j < anArray[i].length; ++j) {
                    System.out.print(anArray[i][j]+"\t");
                }
                System.out.print("\n");
            }

    }

}
```

Using for Loop

```
public class MultidimensionalArrayDemo {

    public static void main(String[] args) {
        // create a 2d array
        int[][] anArray = {
            {1, -2, 3},
            {-4, -5, 6, 9},
            {7},
        };

        // first for...each loop access the individual array
        // inside the 2d array
        for (int[] innerArray: anArray) {
            // second for...each loop access each element inside the row
            for(int element: innerArray) {
                System.out.print(element+"\t");
            }
            System.out.print("\n");
        }

    }
```

Using for each Loop

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Copying Arrays

- The System class has an arraycopy method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src, int srcPos,
                             Object dest, int destPos, int length)
```

```
public class ArrayCopyDemo {

    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                  'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));

    }

}
```

Output: caffein

Sort Arrays

```
import java.util.Arrays;

public class SortArrayDemo {

    public static void main(String[] args) {
        int intArr[] = { 10, 20, 15, 22, 35 };

        Arrays.sort(intArr);
        System.out.println("The sorted Array is:");
        for(int element:intArr) {
            System.out.print(element+" ");
        }

    }

}
```

Output

```
The sorted Array is:
10 15 20 22 35
```


Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Search in an Array

```
import java.util.Arrays;

public class SearchArrayDemo {

    public static void main(String[] args) {
        int intArr[] = { 10, 20, 15, 22, 35 };

        //Binary Search requires Arrays to be sorted first
        Arrays.sort(intArr);

      //Element to search
        int intKey = 22;

        System.out.println(intKey + " found at index = "
                           + Arrays.binarySearch(intArr, intKey));
    }

}
```

Output

22 found at index = 3

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

More about Java Array

- To Learn More about Java Arrays Please refer the following Oracle Documentation

https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Vector Class in Java

Vector Class in Java

- The Vector class implements a growable array of objects.
- Like an array, it contains components that can be accessed using an integer index.
- The size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.
- It is found in the java.util package and implements the List interface, so we can use all the methods of List interface.

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Creating a Vector

- Here is how we can create vectors in Java

```
Vector<Type> vector = new Vector<>();
```

- For example

```
// create Integer type linked list
Vector<Integer> vector= new Vector<>();

// create String type linked list
Vector<String> vector= new Vector<>();
```


Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

Add Elements to Vector

- `add(element)` - adds an element to vectors
- `add(index, element)` - adds an element to the specified position
- `addAll(vector)` - adds all elements of a vector to another vector

```
import java.util.Vector;

public class VectorAddElementsDemo {

    public static void main(String[] args) {
        Vector<String> mammals= new Vector<>();

        // Using the add() method
        mammals.add("Dog");
        mammals.add("Horse");

        // Using index number
        mammals.add(2, "Cat");
        System.out.println("Vector: " + mammals);

        // Using addAll()
        Vector<String> animals = new Vector<>();
        animals.add("Crocodile");

        animals.addAll(mammals);
        System.out.println("New Vector: " + animals);

    }

}
```

Output

```
Vector: [Dog, Horse, Cat]
New Vector: [Crocodile, Dog, Horse, Cat]
```

Access Vector Elements

- `get(index)` – returns an element specified by the index
- `iterator()` – returns an iterator object to sequentially access vector elements

```java
import java.util.Iterator;
import java.util.Vector;

public class VectorAccessElementDemo {

    public static void main(String[] args) {
        Vector<String> animals= new Vector<>();
        animals.add("Dog");
        animals.add("Horse");
        animals.add("Cat");

        // Using get()
        String element = animals.get(2);
        System.out.println("Element at index 2: " + element);

        // Using iterator()
        Iterator<String> iterate = animals.iterator();
        System.out.print("Vector: ");
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }//end of while

    }//end of main

}//end of class
```

Output

```
Element at index 2: Cat
Vector: Dog, Horse, Cat,
```

Remove Vector Elements

- `remove(index)` – removes an element from specified position
- `removeAll()` – removes all the elements
- `clear()` – removes all elements. It is more efficient than `removeAll()`

```
import java.util.Vector;

public class VectorRemoveElementDemo {

    public static void main(String[] args) {
        Vector<String> animals= new Vector<>();
        animals.add("Dog");
        animals.add("Horse");
        animals.add("Cat");

        System.out.println("Initial Vector: " + animals);

        // Using remove()
        String element = animals.remove(1);
        System.out.println("Removed Element: " + element);
        System.out.println("New Vector: " + animals);

        // Using clear()
        animals.clear();
        System.out.println("Vector after clear(): " + animals);

    }

}
```

Output

```
Initial Vector: [Dog, Horse, Cat]
Removed Element: Horse
New Vector: [Dog, Cat]
Vector after clear(): []
```

More about Java Vector Class

- To Learn More about Java Vector Class Please refer the following Oracle Documentation

https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html

Prof. Sarju S, Department of Computer Science and Engineering, SJCET Palai

References

- https://docs.oracle.com/javase/tutorial/
- https://www.w3schools.com/java/java_arrays.asp
- https://www.programiz.com/java-programming/multidimensional-array
- https://beginnersbook.com/2013/05/java-arrays/
- https://www.programiz.com/java-programming/vector

Thank You

Prof. Sarju S
Department of Computer Science and Engineering
St. Joseph's College of Engineering and Technology, Palai
sarju.s@sjcetpalai.ac.in

CST 205 Object Oriented Programming using Java

CST 281 Object Oriented Programming
(As per KTU 2019 Syllabus)

Module 2 - Lecture 2

Module 2 - Core Java Fundamentals

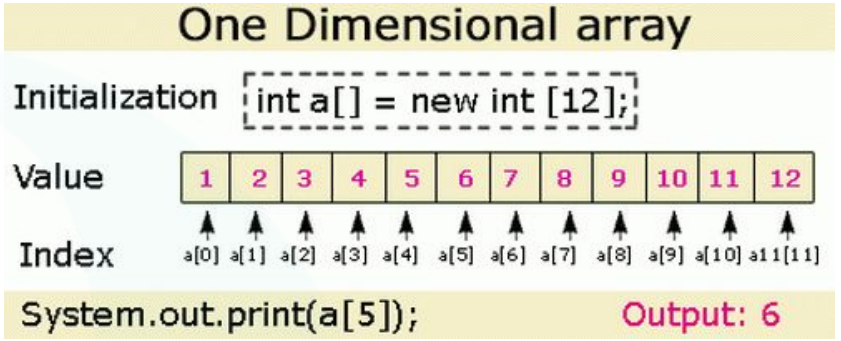
Primitive Data types - Integers, Floating Point Types, Characters, Boolean. Literals, Type Conversion and Casting, Variables, **Arrays, Strings, Vector class**. **Operators** - Arithmetic Operators, Bitwise Operators, Relational Operators, Boolean Logical Operators, Assignment Operator, Conditional (Ternary) Operator, Operator Precedence. **Control Statements** - Selection Statements, Iteration Statements and Jump Statements.

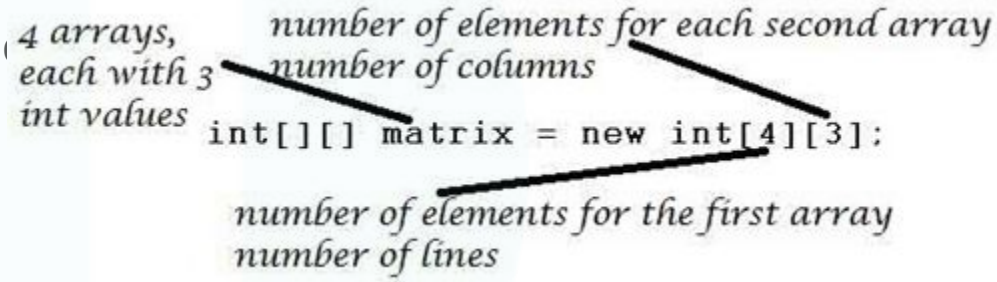
Object Oriented Programming in Java - Class Fundamentals, Declaring Objects, Object Reference, Introduction to Methods, Constructors, this Keyword, Method Overloading, Using Objects as Parameters, Returning Objects, Recursion, Access Control, Static Members, Final Variables, Inner Classes, Command Line Arguments, Variable Length Arguments.

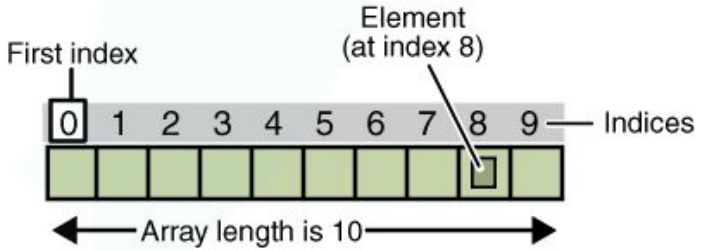
Inheritance - Super Class, Sub Class, The Keyword super, protected Members, Calling Order of Constructors, Method Overriding, the Object class, Abstract Classes and Methods, using final with Inheritance.

Arrays in Java

- Arrays are used to store multiple values of same type, in a single variable
- Arrays are objects which store multiple values of the **same type**
- It can hold primitive types as well as references.
- Features of Array
- Arrays are objects
- They can even hold the reference variables of other objects
- They are created during **runtime** (new)
- They are dynamic, created on the **heap**
- The Array length is fixed.

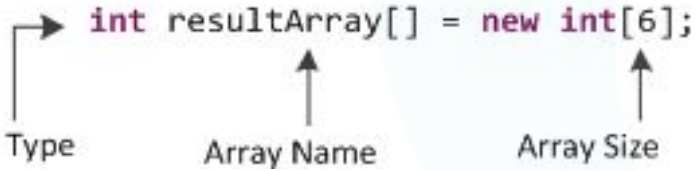






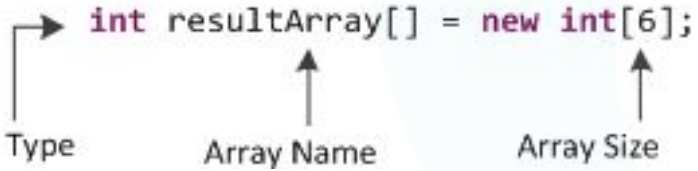
Array declaration

- To declare an array, define the variable type with square brackets
- Eg: int[] anArray;
- You can also place the brackets after the array's name:
- // this form is discouraged
- float anArrayOfFloats[];
- However, convention discourages this form; the brackets identify the array type and should appear with the type designation.



Array declaration

- To declare an array, define the variable type with square brackets
- Eg: int[] anArray; // Creates array reference
- You can also place the brackets after the array's name:
- // this form is discouraged
- float anArrayOfFloats[]; // Creates array reference
- However, convention discourages this form; the brackets identify the array type and should appear with the type designation.



Definition, Initializing and Accessing arrays

One way to create an array is with the new operator.

```
// declares an array of integers
int [] anArray // create an array of integers
anArray = new int [10] ;
```

To assign values to each element of the array

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // and so forth
```

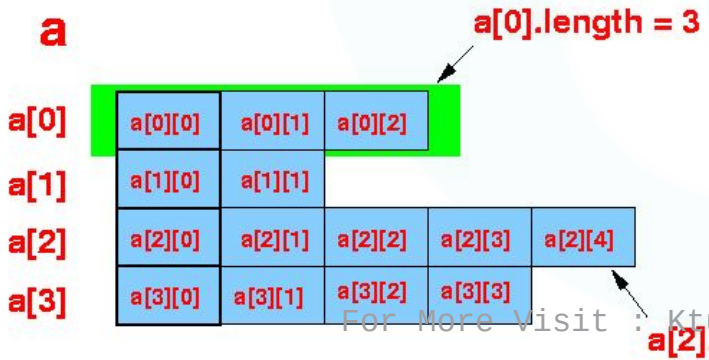
Each array element is accessed by its numerical index

```
System.out.println("Element 1 at index 0: " + anArray [0]);
System.out.println("Element 2 at index 1: " + anArray [1]);
System.out.println("Element 3 at index 2: " + anArray [2]);
```

Array length

Alternatively, you can use the shortcut syntax to create and initialize an array
int[] anArray = {100, 200, 300};
Here the length of the array is determined by the number of values provided between braces and separated by commas.

To find out how many elements an array has, use the length property:
String[] cars = {"Volvo", "BMW", "Ford"};
System.out.println(cars.length);// Outputs 3



Multidimensional Arrays

- A multidimensional array is an array of arrays. Each element of a multidimensional array is an array itself. For example int[][] anArray = new int [3][4];
- To create a two dimensional array, add each array within its own set of curly braces:
- int [][] anArray ={ {1, 2, 3}, {4, 5, 6, 9} };
- //calculate the length of each row
- System.out.println("Length of row 1: " + anArray [0].length);
- System.out.println("Length of row 2: " + anArray [1].length);

Iterating array elements

- We can use for...each loop or for loop to access elements of the multidimensional array.

```
public class MultidimensionalArrayDemo1 {

    public static void main(String[] args) {
        int[][] anArray = {
                {1, -2, 3},
                {-4, -5, 6, 9},
                {7},
            };

            for (int i = 0; i < anArray.length; ++i) {
                for(int j = 0; j < anArray[i].length; ++j) {
                    System.out.print(anArray[i][j]+"\t");
                }
                System.out.print("\n");
            }

    }

}
```

```
public class MultidimensionalArrayDemo {

    public static void main(String[] args) {
        // create a 2d array
        int[][] anArray = {
            {1, -2, 3},
            {-4, -5, 6, 9},
            {7},
        };

        // first for...each loop access the individual array
        // inside the 2d array
        for (int[] innerArray: anArray) {
            // second for...each loop access each element inside the row
            for(int element: innerArray) {
                System.out.print(element+"\t");
            }
            System.out.print("\n");
        }

    }
```

Copying array

- The System class has an arraycopy method that you can use to efficiently copy data from one array into another

public static void arraycopy (Object src , int srcPos, Object dest , int destPos , int length) ;

```
public class ArrayCopyDemo {

    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));

    }

}
```

Output: caffein

Sorting array

```
import java.util.Arrays;

public class SortArrayDemo {

    public static void main(String[] args) {
        int intArr[] = { 10, 20, 15, 22, 35 };

        Arrays.sort(intArr);
        System.out.println("The sorted Array is:");
        for(int element:intArr) {
            System.out.print(element+" ");
        }

    }

}
```

Output

```
The sorted Array is:
10 15 20 22 35
```

Searching array

```
import java.util.Arrays;

public class SearchArrayDemo {

    public static void main(String[] args) {
        int intArr[] = { 10, 20, 15, 22, 35 };

        //Binary Search requires Arrays to be sorted first
        Arrays.sort(intArr);

      //Element to search
        int intKey = 22;

        System.out.println(intKey + " found at index = "
                           + Arrays.binarySearch(intArr, intKey));
    }

}
```

Output

22 found at index = 3

Strings

- In Java, string is basically an object of String class, that represents sequence of char values. An array of characters works same as Java string
- Strings are treated as objects of class String which represents character strings.
- All string literals in Java programs, such as "abc", are implemented as instances of this class.

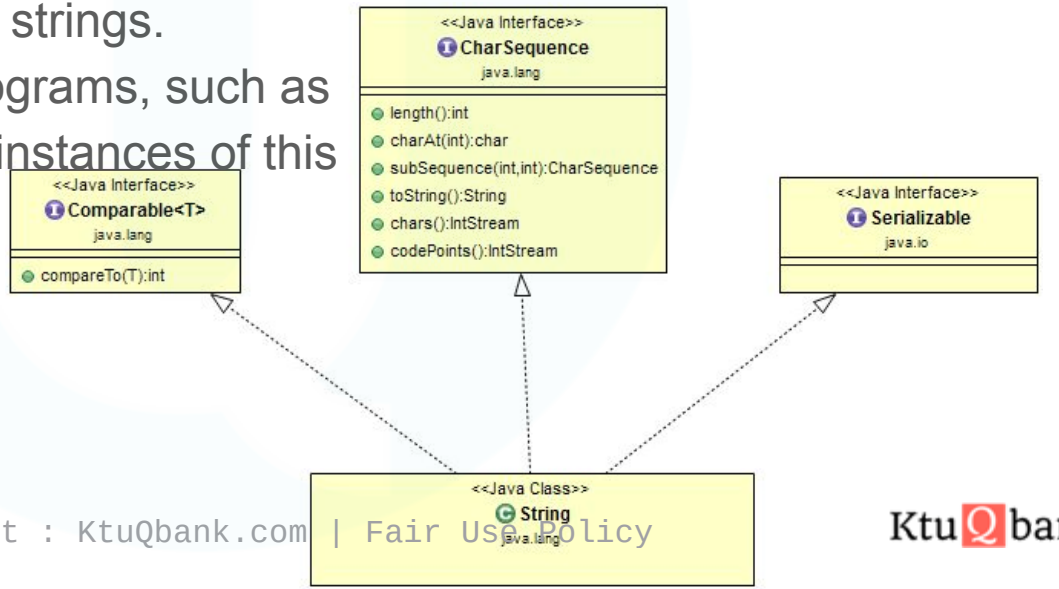
java.lang

Class String

java.lang.Object
    java.lang.String

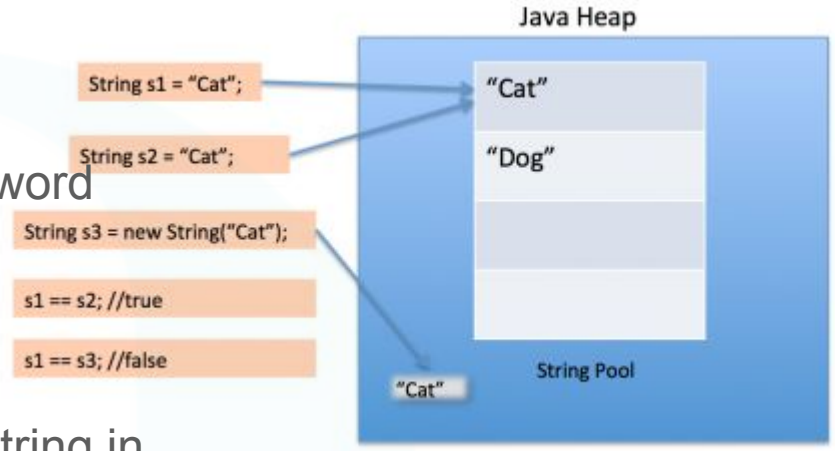
All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>



char[] ch={'h','e','l','l','o'};
String s=new String(ch);
This is same as that of
String s="hello";

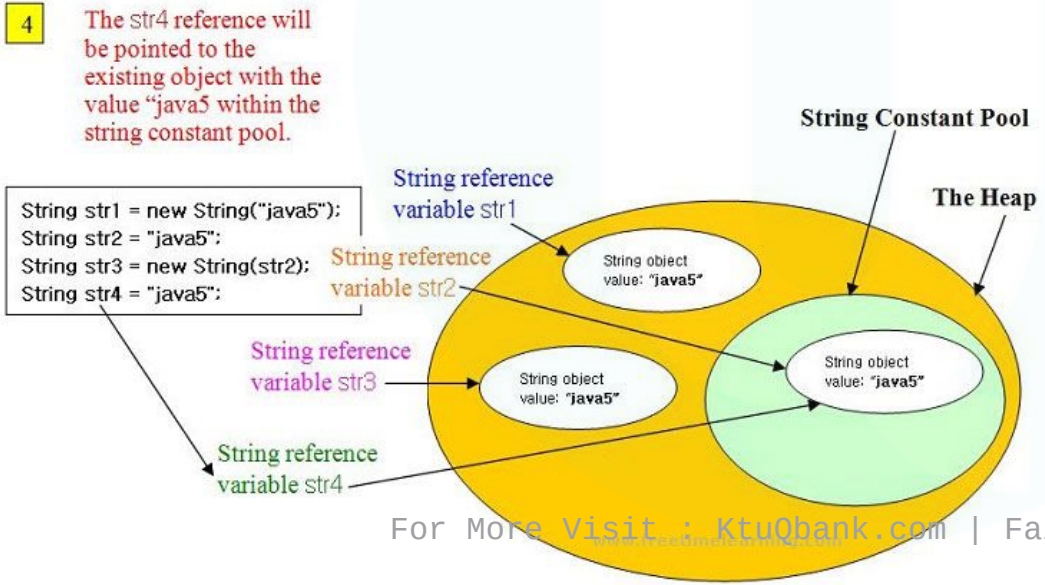
String literals vs objects



- String objects are created using **new** keyword
- String s1 = new String("Java");
- String s2 = new String("Java");
- System.out.println(s1 == s2); //**false**
- String literals are created by placing the string in double quotes.
- String s3 = "Java";
- String s4 = "Java";
- System.out.println(s3 == s4);//**true**
- Here **==** (equality) is not checking the content of the stings.
- String literals are stored in a special area called **String Constant Pool** in program heap.

Immutability of string

- Strings are constant; their values cannot be changed(**immutable**) after they are created. **StringBuffer** support mutable strings.
- String objects are immutable, so they can be shared.
- In Java, String is a final and immutable class.
- It cannot be inherited, and once created, we can not alter the object.

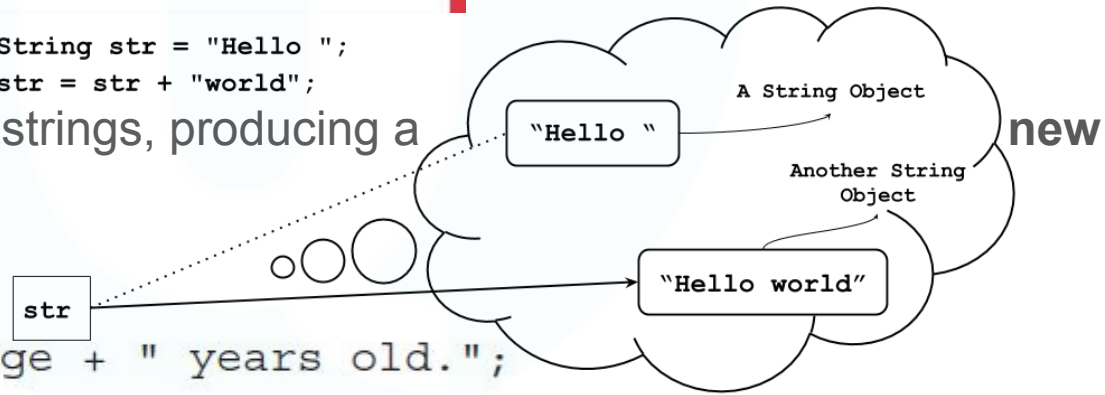


String length and String concatenation

- The length of a string is the number of characters that it contains.
- To obtain this value, call the length( ) method.

```
String str = "This is a string";
int length = str.length();
System.out.println (length);
```

```
String str = "Hello ";
str = str + "world";
```



+ operator concatenates two strings, producing a **new** String object as the

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

String comparison

- equals( ) and equalsIgnoreCase()
- These functions produces boolean output.
- It has this general form: boolean equals(Object str)

```
String s1 = "Hello";
String s2 = "Hello";
s1.equals(s2); // true

String s1 = "Hello";
String s2 = "HELLO" ;
s1.equals(s2); // false
s1.equalsIgnoreCase(s2); // true
```

String comparison

```
String s1="java string split method";
System.out.println(s1.startsWith("ja")); // true
System.out.println(s1.startsWith("java string")); // true
```

startsWith( ) and endsWith ()

- The startsWith( ) method determines whether a given String begins with a specified string.
- endsWith( ) determines whether the String in question ends with a specified string.
- They have the following general forms
  - boolean startsWith (String str)
  - boolean endsWith (String str)

```
String s1="java string split method";
System.out.println(s1. endsWith("d")); // true
System.out.println(s1. endsWith(" method ")); // true
```

String comparison: compareTo() method

- To find less than, equal to, or greater than relations between strings
- A string is less than another if it comes before the other in dictionary order
- A string is greater than another if it comes after the other in dictionary order

int compareTo(String *str*)

Here, *str* is the **String** being compared with the invoking **String**. The result of the comparison is returned and is interpreted as shown here:

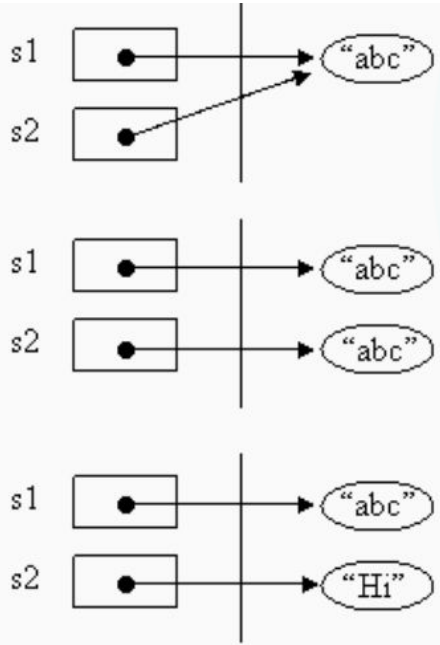
Value	Meaning
Less than zero	The invoking string is less than *str*.
Greater than zero	The invoking string is greater than *str*.
Zero	The two strings are equal.

```
String s1="hello";
String s2="hello";
String s3="meklo";
String s4="hemlo";
String s5="flag";
System.out.println(s1.compareTo(s2));//0 because both are equal
System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than "m"
System.out.println(s1.compareTo(s4));//-1 because "l" is 1 times lower than "m"
System.out.println(s1.compareTo(s5));//2 because "h" is 2 times greater than "f"
```

String comparison: equals() vs ==

- equals( ) method compares the characters inside a String object.
- == operator compares two object references to see whether they refer to the same instance

```
String s1 = new String("HELLO");
String s2 = new String("HELLO");
System.out.println(s1 == s2); // false
System.out.println(s1.equals(s2));// true
```



equals() vs == in Java String

String searching

- The String class provides two methods that allow you to search a string for a specified character or substring
- indexOf( ) Searches for the first occurrence of a character or

```
String s1="this is index of example";
//passing substring
int index1=s1.indexOf("is");//returns the index of is substring
int index2=s1.indexOf("index");//returns the index of index substring
System.out.println(index1+"  "+index2);
```

Output: 2 8

```
String s1="this is index of example";//there are 2 's' characters in this Sentence
int index1=s1.lastIndexOf('s');//returns last index of 's' char value
System.out.println(index1);//6
```

substring() method

- Used to get the substring of a given string based on the passed indexes
  - String substring(int beginIndex)
  - Returns the substring starting from the specified index i.e beginIndex and extends to the character present at the end of the string
- String substring(int beginIndex , int endIndex
  - The substring begins at the specified beginIndex and extends to the character at index endIndex - 1.
  - Thus the length of the substring is endIndex - beginIndex .

```
public class SubStringExample{
   public static void main(String args[]) {
       String str= new String("quick brown fox jumps over the lazy dog");
       System.out.println("Substring starting from index 15:");
       System.out.println(str.substring(15));
       System.out.println("Substring starting from index 15 and ending at 20:");
       System.out.println(str.substring(15, 20));
   }
}
```

String methods

char	charAt(int index) Returns the char value at the specified index	char first = str.charAt (0)
boolean	equals(Object object) Compares string to another string	bool isEqual = str.equals (other)
boolean	equalsIgnoreCase(String other) Compares to another string, ignoring case	bool isEqual = str. equalsIgnoreCase(other)
int	indexOf(String str) Returns the index of str in the given string	int index = str.indexOf (“Hello”)
int	length()	int length = str.length()
String	substring(int beginIndex)	int rest = str.substring (5)
String	substring(int beginIndex, int endIndex)	int firstTwo = str. substring(0, 2)

Vector Class in Java

- Vector implements a dynamic array, ie growable array of objects.
- Like an array, it contains elements that can be accessed using an integer index .
- It is found in the java.util package and implements the List interface , so we can use all the methods of List interface.
- It is synchronized and due to which it gives poor performance in searching, adding, delete and update of its elements.
- Contains many legacy methods that are not part of the collections 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.

Creating a Vector

java.util

Class Vector<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.Vector<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

Stack

Here is how we can create vectors in Java

- Import Vector class. (import java.util.Vector ;)

Vector<Type> vector = new Vector<>();

For example

// create Integer type linked list

Vector<Integer> vector= new Vector<>();

// create String type linked list

Vector<String> vector= new Vector<>();

Add elements to vector

- `add(element)` - adds an element to vectors
- `add(index, element)` - adds an element to the specified position
- `addAll(vector)` - adds all elements of a vector to another vector

```
import java.util.Vector;

public class VectorAddElementsDemo {

    public static void main(String[] args) {
        Vector<String> mammals= new Vector<>();

        // Using the add() method
        mammals.add("Dog");
        mammals.add("Horse");

        // Using index number
        mammals.add(2, "Cat");
        System.out.println("Vector: " + mammals);

        // Using addAll()
        Vector<String> animals = new Vector<>();
        animals.add("Crocodile");

        animals.addAll(mammals);
        System.out.println("New Vector: " + animals);

    }
}
```

Output

```
Vector: [Dog, Horse, Cat]
New Vector: [Crocodile, Dog, Horse, Cat]
```

Access vector elements

- `get(index)` – returns an element specified by the index
- `iterator()` – returns an iterator object to sequentially access vector elements

```
import java.util.Iterator;
import java.util.Vector;

public class VectorAccessElementDemo {

    public static void main(String[] args) {
        Vector<String> animals= new Vector<>();
        animals.add("Dog");
        animals.add("Horse");
        animals.add("Cat");

        // Using get()
        String element = animals.get(2);
        System.out.println("Element at index 2: " + element);

        // Using iterator()
        Iterator<String> iterate = animals.iterator();
        System.out.print("Vector: ");
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }//end of while

    }//end of main

}//end of class
```

Output

```
Element at index 2: Cat
Vector: Dog, Horse, Cat,
```

Remove vector elements

- `remove(index)` – removes an element from specified position
- `removeAll()` – removes all the elements
- `clear()` – removes all elements. It is more efficient than `removeAll()`

```
import java.util.Vector;

public class VectorRemoveElementDemo {

    public static void main(String[] args) {
        Vector<String> animals= new Vector<>();
        animals.add("Dog");
        animals.add("Horse");
        animals.add("Cat");

        System.out.println("Initial Vector: " + animals);

        // Using remove()
        String element = animals.remove(1);
        System.out.println("Removed Element: " + element);
        System.out.println("New Vector: " + animals);

        // Using clear()
        animals.clear();
        System.out.println("Vector after clear(): " + animals);

    }

}
```

Output

```
Initial Vector: [Dog, Horse, Cat]
Removed Element: Horse
New Vector: [Dog, Cat]
Vector after clear(): []
```

Important Vector methods

Method	Description
v.add(o)	adds Object o to Vector v
v.add(i, o)	Inserts Object o at index i, shifting elements up as necessary.
v.clear()	removes all elements from Vector v
v.contains(o)	Returns true if Vector v contains Object o
v.firstElement(i)	Returns the first element.
v.get(i)	Returns the object at int index i.
v.lastElement(i)	Returns the last element.
v.size()	Returns the number of elements in Vector v.

References

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.
- Visual Paradigm - www.visual-paradigm.com
- GeeksforGeeks - www.geeksforgeeks.org
- Wikipedia - www.wikipedia.org
- Tutorialspoint - www.tutorialspoint.com
- Java Zone - https://dzone.com

Topics

- Core Java Fundamentals:

✓ **Object Oriented Programming in Java**

- **Class Fundamentals**
- **Declaring Objects**
- **Object Reference**
- **Introduction to Methods.**

Class Fundamentals

- The **class** is the core of Java.
  - The class forms the basis for object-oriented programming in Java.
- A class is a "blueprint" for creating objects
- A **class** is a ***template for an object.***
  - An **object** is an ***instance of a class.***
- A class defines a *new type of data*.
- A class creates a ***logical framework*** that defines the relationship between its members.

Example

Student

(class)

Rollno Name
read() write()

properties
(instance variables)

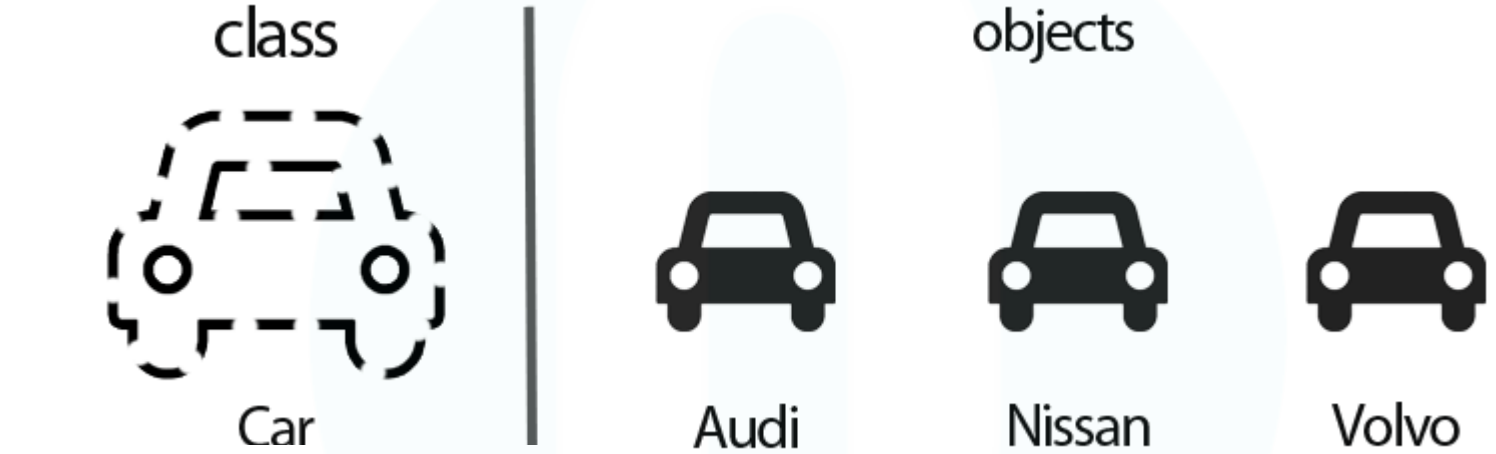
behaviour
(methods)

12
Smith

object

2
Susan

object



model colour price	**properties** (instance variables)
start() brake() reverse() stop()	**behaviour** (methods)

Class(continued.)

- A class is declared using the keyword **class**
- The data or variables, defined within a class are called **instance variables**.
  - because each instance of the class (that is, each object of the class) contains its own copy of these variables.
  - the data for each object is separate and unique.
- Functions inside class are called **methods**.
- The methods and variables defined within a class are called **members of the class.**

The General Form of a Class

- **A** general form of a **class definition is**

class *classname*

{

type ***instance-variable1;***

type ***instance-variableN;***

type methodname1(parameter-list)

{

// body of method

}

type methodnameN(parameter-list)

{ // body of method

}

}

Properties **(instance variables)**

Behaviour **or Method** or **function**

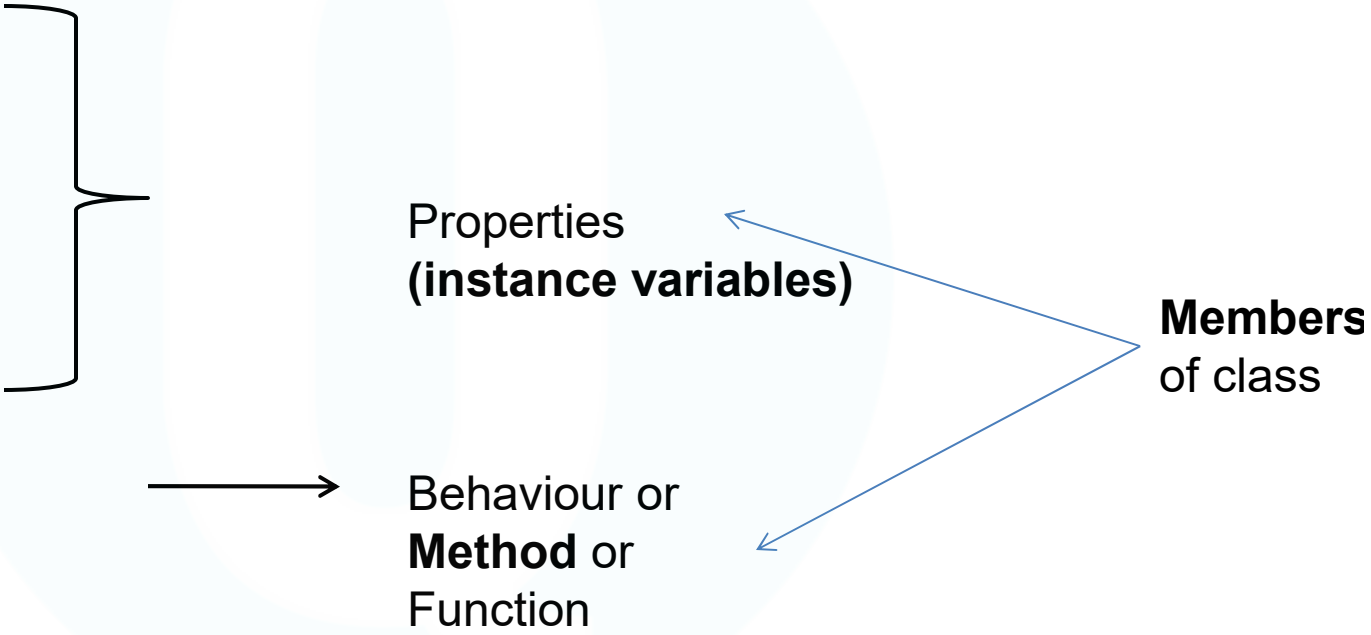
Behaviour or **Method** or **function**

Members of class

A Simple Class

A Simple Class

```
class Box
{
double width;
double height;
double depth;

void  volume()
{
//statements
}

}
```



Declaring Objects

- When we create a class, we are creating a new data type.
  - We can use this type to declare objects of that type.
- Obtaining objects of a class is a two-step process.
  - *First*, we must **declare a variable of the class type.**
    - This variable does not define an object.
    - It is simply a variable that can *refer to an object.*
  - *Second*, we must **acquire an actual, physical copy of the object and assign it to that variable** (using **new** operator)

Declaring Objects(contd.)


```
Classname objectname ;          // declare reference to object
objectname = new Classname(); // allocate an object
```

- We can write this in a single statement

```
Classname objectname = new Classname();
```

```
class Box
{double Width;
double Height;
double Depth;
}
Box mybox;
```

- This line declares **mybox** as a **reference to** an object of type **Box**.
- Here **mybox** contains the value **null**, which indicates that it does not yet point to an actual object

```
mybox = new Box();
```

- This line allocates an actual object and assigns a reference to it to **mybox.**
- **mybox** holds the memory address of the actual Box object.

Declaring Objects(contd.)

```
class Box
{double Width;
double Height;
double Depth;
}
```

Declaring an object of type **Box**

Statement	Effect
Box mybox;	null mybox
mybox = new Box();	mybox → Width, Height, Depth Box object

Declaring Objects(contd.)

- The class name followed by parentheses specifies the *constructor* for the class.

Box mybox=new Box();

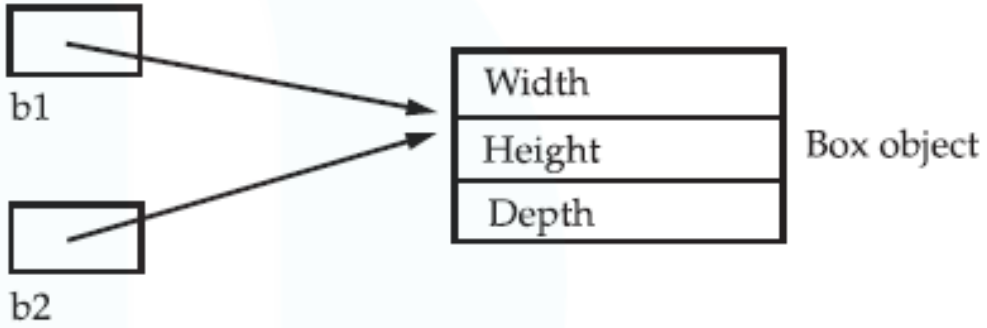
- Here Box is the class. Box()is the constructor.
- A constructor defines what occurs when an object of a class is created.

Assigning Object Reference Variables

- Object reference variables act differently when an assignment takes place
- E.g.

Box b1 = new Box();

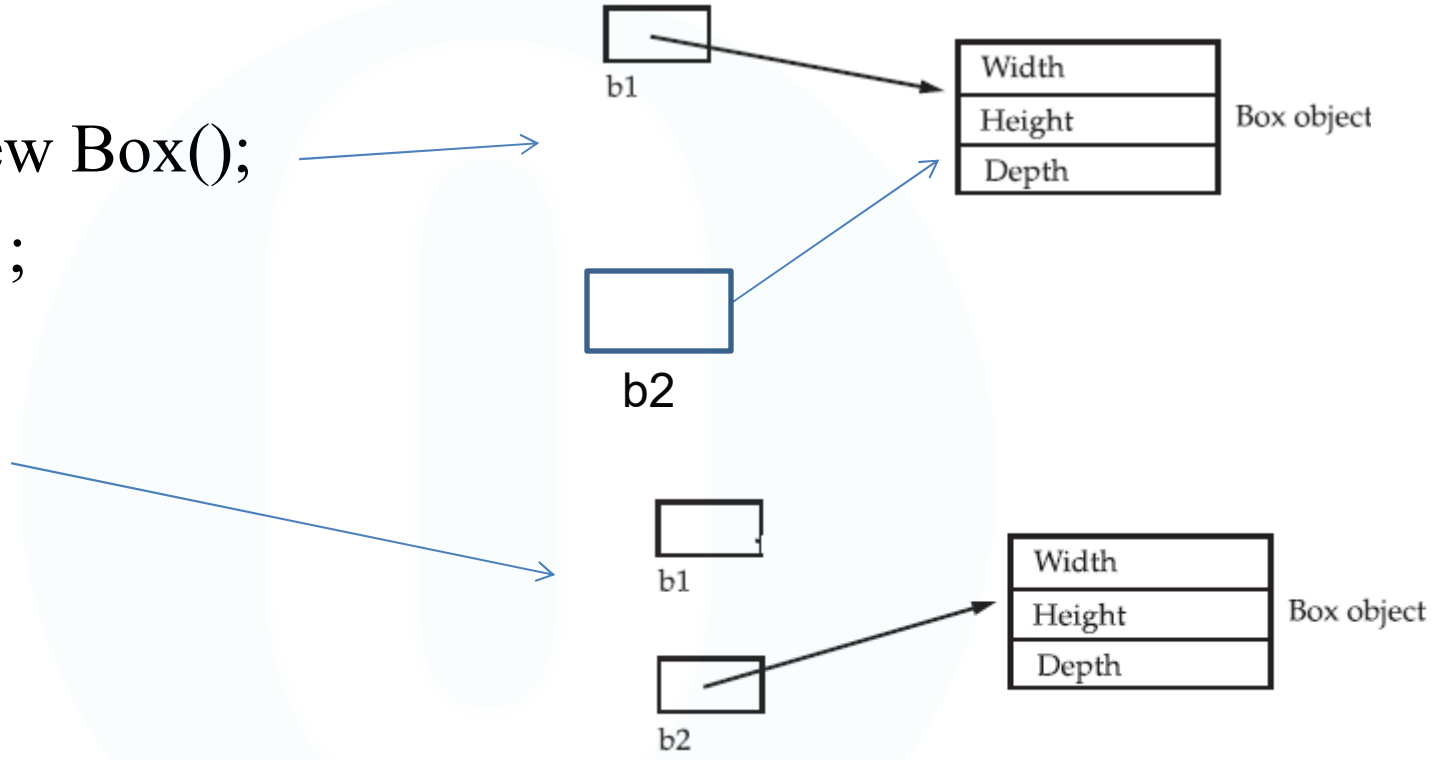
Box b2 = b1;



- Here b1 and b2 will both refer to the ***same object.***
- Any changes made to the object through b2 will affect the object which is referred by b1, because they are the same object.

Assigning Object Reference Variables (contd.)

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```



- Here at the end b1 has been set to null, but b2 still points to the original object.

Class vs object

Class	Object
• **Template** for creating objects	• **Instance** of class
• **Logical** entity	• **Physical** entity
• Declared using **class** keyword	• Created using **new** operator.
• Class **does not get any memory** when it is created.	• Object **gets memory** when it is created using new operator.
• A class is **declared only once**	• **Many objects** can be created from a class

Introducing Methods

- **Classes** usually consist of two things:
  - Instance variables
  - Methods or functions.
- The general form of a method:

type name(parameter-list)

{

// body of method

}

- The *type* specifies the type of data returned by the method.
  - any valid type, including class types, void
- The *parameter-list or argument list is a* sequence of type and identifier pairs separated by commas.

Introducing Methods(contd.)

- Methods that have a return type other than void return a value to the calling routine using the following form of the return statement:

  return value;

- Method of one class can be invoked by functions of other classes through objects of former class.

Objectname.method(parameters);

```
// EXAMPLE
class Box {
        double width;
        double length;
        double depth;

        void volume()
        {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
        }
}
```

Properties
(instance variables)

Behaviour or
Method or
Function

Box class

```
class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box();
mybox1.width = 10;
mybox1. length = 30;
mybox1.depth = 15;
mybox1.volume();
} }
```

Behaviour or
Method or
Function
MAIN FUNCTION

Object of class **Box**

BoxDemo class

Example

- Create a class Box with instance variables length, width and height. Include a method volume to compute the volume of the box,
- Create another class BoxDemo with main function that creates an object of class Box named mybox1 and set the values for instance variables(length, width and height). Invoke the function volume in Box to compute the volume of the created object mybox1

```
class Box {
        double width;
        double length;
        double depth;

        void volume()
        {
        System.out.print("Volume is ");
        System.out.println(width * length * depth);
        }
}
class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box();
mybox1.width = 10;
mybox1. length = 30;
mybox1.depth = 15;
mybox1.volume();
}
}
```

OUTPUT

Volume is 3000.0

```
// program using return statement
class Box {
        double width;
        double height;
        double depth;

        void volume()
        {
        return(width * length * depth);
        }
}
class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box();
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
int v=mybox1.volume();
System.out.println(”Volume=“+v);
} }
```

OUTPUT

Volume is 3000.0

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

Topics

- Core Java Fundamentals:

✓ **Constructors**

✓ **this Keyword**

✓ **Method Overloading**

✓ **Using Objects as Parameters**

Constructor

- A constructor **help to initialize an object**(give values) immediately upon creation.
- Constructor is a special method inside the class.
- Constructor has the same name as the class in which it resides.
- Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.

Constructor(contd.)

- Constructors have no return type, not even void.
  - This is because the implicit return type of a class' constructor is the class type itself.
- Two types of constructors
  - Default constructor – has no arguments
  - Parameterized constructor –has arguments(parameters)

Constructor(contd.)

- **Default constructor** has no arguments or parameters.

E.g.

```
class A
{
    A( )
    {
    //statements
    }
}
```

Default constructor(contd.)


```
class Box
{
int width ,length,height;
Box()
{
width=10;
length=10;
height=10;
}}
```

The following statement creates an object of class Box.

Box mybox1 = **new Box();**

Here **new Box( )** is calling the **Box( )** constructor**.**

Default constructor(contd.)


```
class Box
{
int width ,length,height;
Box()
{
width=10;
length=10;
height=10;
}}
```

The following statement creates an object of class Box.

Box mybox1 = **new Box();**

Here **new Box( )** is calling the **Box( )** constructor**.**

Default constructor(contd.)

- When we do not explicitly define a constructor for a class, then **Java creates a default constructor for the class.**

```
class Box {
    int width;
    int length;
    int height;
    Box()
    {
    System.out.println("Constructing Box");
    width = 10;
    length = 10;
    height= 10;
    }
    int volume()
    {
    return width * length * height;
    }
}
```

```
class Box {
        int length;
        int height;
        int width;
        Box()
        {
        System.out.println(“Constructor");
        width = 10;
        length = 10;
        height= 10;
        }
        int volume()
        {
        return width * length * height;
        }
}
```

```
class BoxDemo {
public static void main(String args[])
{
Box mybox1 = new Box();
Box mybox2 = new Box();
int vol;
vol = mybox1.volume();
System.out.println("Volume is " + vol);

vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

OUTPUT

```
Constructor
Constructor
Volume is 1000
Volume is 1000
```

Parameterized Constructors

- Constructors with arguments are called parameterized constructors.

```
class Box
{
double width;
double height;
double length;
Box(double w, double h, double l)
{
width = w;
height = h;
length= l;
}
double volume()
{
return width * height * length;
}
}
```

Parameterized Constructor of class Box (Box constructor has arguments-> parameters)

```
class Box
{
double width;
double height;
double length;
Box(double w, double h, double l)
{
width = w;
height = h;
length = l;
}

double volume()
{
return width * height * length;
}
}
```

```
class BoxDemo {
public static void main(String args[]) {

Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 2);
double vol;
vol = mybox1.volume();
System.out.println("Volume is " + vol);

vol = mybox2.volume();
System.out.println("Volume is " + vol);
 }
}
```

OUTPUT
Volume is 3000
Volume is 36

Parameterized constructor(contd.)

Box mybox1 = new **Box(**10, 20, 15);

- Here the values 10, 20, and 15 are passed to the **Box( )** constructor when new creates the object mybox1.
- The parameterized constructor is

```
Box(double w, double h, double l)
{
width = w;
height = h;
length = l;
}
```

- Thus, value of mybox1 object's width, height, and depth will be set as 10, 20, and 15 respectively.

The this Keyword

- The **this** keyword can be used inside any method to refer to the **current object**.
- **this** is always a reference to the object on which the method was invoked.
- **this** can be used to refer current class instance variable.
- **this** can be used to invoke current class method (implicitly)
- **this**() can be used to invoke current class constructor.
- **this** can be passed as an argument in the method call.
- **this** can be passed as argument in the constructor call.

this-Example

```
Box(double w, double h, double l)
 {
this.width = w;
this.height = h;
this.length = l;
}
```

Here **this** will always refer to the object invoking the method

```
class Box
{
double width;
double length;
double height;
Box(double w, double l, double h)
{
this.width = w;
this.length = l;
this.height = h;
}
}
```

```
class BoxDemo {
public static void main(String args[]) {

Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 2);
}
}
```

Here in statement
Box mybox1 = new Box(10, 20, 15);
mybox1 object is created by calling parameterized constructor.
Box(double w, double l, double d)
Here **this** inside constructor refers to object mybox1.

Next when **mybox2** object is created, **this** refers to object mybox2.

Instance variable hiding-using this

- We can have **local variables**, including formal parameters to methods, which has the same name of the class' **instance variables(attributes)**.
- But when a local variable has the same name as an instance variable, **the local variable *hides the* instance variable.**
  - **this** helps to solve this. Use **this.** along with instance variables.

Instance variable hiding-using this (contd.)

- // Use **this** to resolve name-space collisions.

```
class Box
{
double width
double length;
double height;
Box(double width, double height, double length)
{
this.width = width;
this.length = length;
this. height; = length;
} }
```

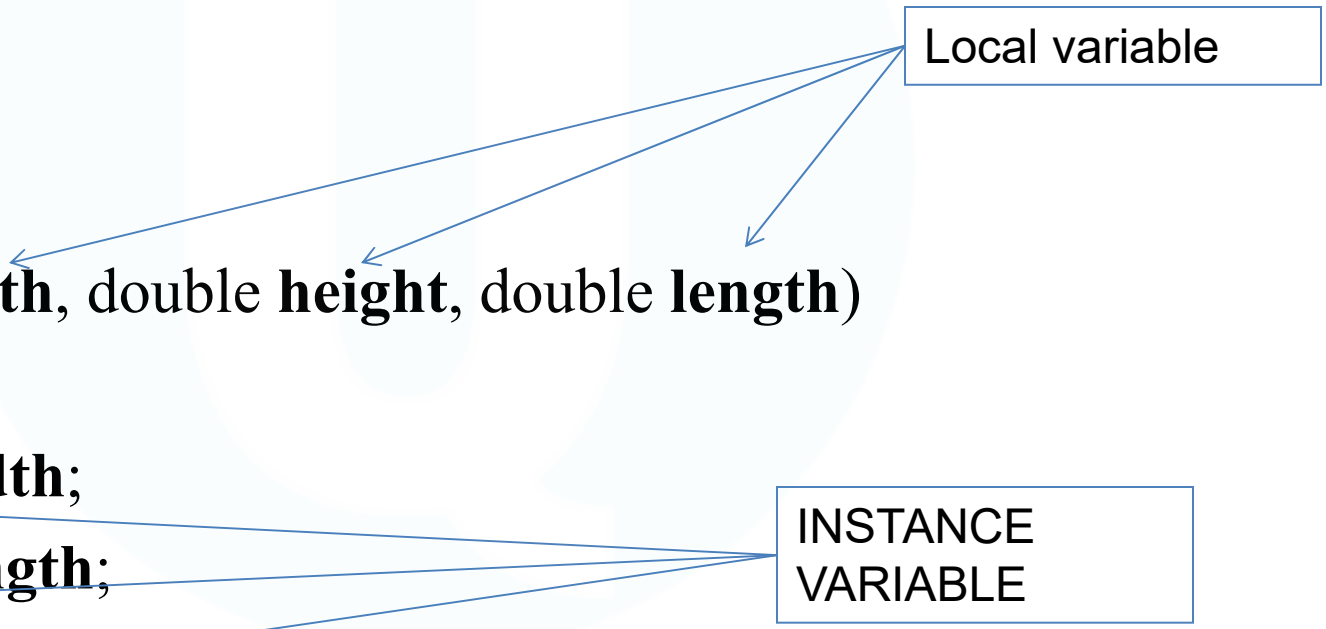
INSTANCE VARIABLE

CONSTRUCTOR

Instance variable hiding-using this (contd.)

- // Use **this** to resolve name-space collisions.

```
class Box
{
double width
double length;
double height
Box(double width, double height, double length)
{
this.width = width;
this.length = length;
this. height = length;
} }
```



Method Overloading

- It is possible to define **two or more methods** with **same name** within the same class, but their parameter declarations should be different.
  - This is called **method overloading.**
  - This is a form of **polymorphism** (many forms)
- Overloaded methods must **differ in the type and/or number of their parameters. (**return types is not significant.)
- When an overloaded method is invoked, Java uses the type and/or number of arguments to determine which version of the overloaded method to actually call.

```
// Demonstrate method
    overloading.
class Over
{
void test()
{
    System.out.println("Empty");
}
void test(int a) {
    System.out.println("a: " + a);
}
void test(int a, int b) {
    System.out.println("a="+a);
    System.out.println("b="+b);
    }
}
```

```
class Sample {
public static void main(String args[])
{
Over ob = new Over();

ob.test();
ob.test(10);
ob.test(2, 5);

}
}
```

OUTPUT
Empty
a=10
a=2
a=5

- *In the example* ,test( ) is overloaded three times.
  - The first version test() takes no parameters,
  - the second **test(int a)**takes one integer parameter
  - the thrd **test(int a,int b)** takes two integer parameters.

Method Overloading(contd.)

- When an overloaded method is called**, Java looks for a match between the arguments** used to call the method and the method's parameters
- This **match need not always be exact**.
  - In some cases, Java's automatic type conversions can play a role in overload resolution.

Overloading -through automatic type conversions


```
class Over{
void test() {
System.out.println("Empty");
}

void test(double a)
 {
System.out.println("a: " + a);
}
}
```

```
class Sample {
public static void main(String
    args[])
{
Over ob = new Over();

ob.test();
ob.test(10);
ob.test(2.5);

}
}
```

OUTPUT
Empty
a=10
a=2.5

Overloading -through automatic type conversions(contd.)

- In this example when **test( ) is called with an integer argument** inside .
  - Overload, no matching method is found with int as argument**.**
- However, Java can automatically **convert an integer into a double,** and this conversion can be used to resolve the call.
  - Therefore, when **test(int) is not found,** Java elevates int to double and then **calls test(double).**

Overloading Constructors

- Constructors can be overloaded. Because a class can have any number of constructors
  - one default constructor, many parameterized constructors

```
class A
{
A() { //statements}
A(int a) { //statements}
A(int a,float b) { //statements}

}
```

```
class Box
{
double width;
double length;
double height;
Box(double w, double l, double h)
{
width = w;
length = l;
height = h;
}
Box()
{
width = 0;
length =0;
height =0;
} }
```


```
class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box(3, 6, 2);
System.out.println("mybox1");
System.out.println(mybox1 .width + " “
    +mybox1 .length + " "+  mybox1 .height);

System.out.println("mybox2");
System.out.println(mybox2.width + " “ +
    mybox2.length + " " +  mybox2 .height);
} }
```

OUTPUT

```
mybox1
0.0   0.0   0.0
mybox2
3.0   6.0   2.0
```

```
class Box
{
double width
double length;
double height;
Box(double w, double l, double h)
{
this.width = w;
this.length = l;
this.height = h;
}
}
```

```
class BoxDemo {
public static void main(String args[]) {

Box mybox1 = new Box(); //ERROR
Box mybox2 = new Box(3, 6, 2);
}
}
```

ERROR

Here following statement tries to create object mybox1 of class Box ,
Box mybox1 = new Box();
This should call default constructor **Box()** in class Box.
But Box class has constructor but no default constructor is there.

So ERROR occurs

```
class Box
{
double width
double length;
double height;
}
```

```
class BoxDemo {
public static void main(String args[]) {

Box mybox1 = new Box();
}
}
```

NO ERROR in this code
The following statement creates object of Box class mybox1
Box mybox1 = new Box();
Since no constructors are not there,
Java provides the default constructor.

Using Objects as Parameters

- We can pas objects as arguments(parameters) to function(method).
- Objects are **passed by reference(call by refernce).**

Object as parameters


```
class Test {
    int a, b;
  Test(int i, int j)
   {
   a = i;
   b = j;
   }
boolean equals(Test o)
 {
if(o.a == a && o.b == b)
 return true;
else return false;
}
}
```

```
class PassOb {
public static void main(String args[])
{
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println(ob1.equals(ob2));
System.out.println(ob1.equals(ob3));
}}
```

OUTPUT

true
false

Object as parameters


```
class Test {
int a, b;
Test(int i, int j)
{
a = i;
b = j;
}
boolean equals(Test o)
 {
if(o.a == this.a && o.b == this.b)
 return true;
else return false;
}
}
```

```
class PassOb {
public static void main(String args[])
{
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println(ob1.equals(ob2));
System.out.println(ob1.equals(ob3));
}}
```

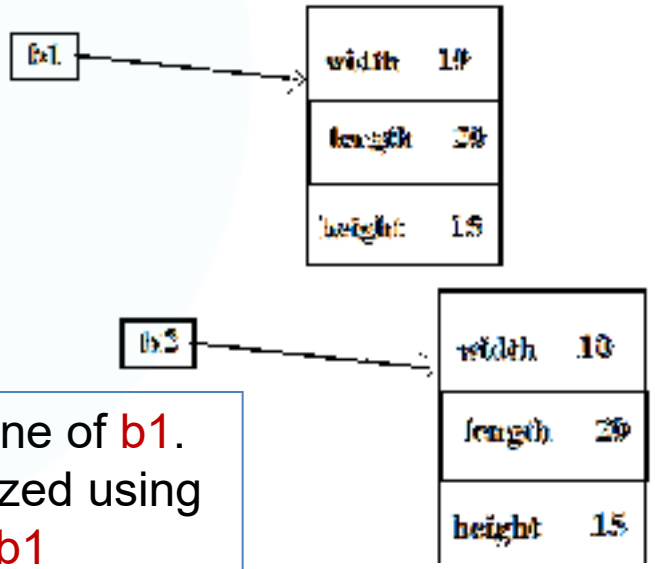
OUTPUT

true
false

Object to initialize another object

```
class Box
{
double width
double length;
double height;
Box(double w, double l, double h)
{
width = w;
length = l;
height = h;
}
}
```

```
class BoxDemo {
public static void main(String args[])
{
Box b1 = new Box(10, 20, 15);
Box b2 = new Box(b1);
}
}
```



Here object b2 is a clone of b1. The object b2 is initialized using initial values of object b1

Passing arguments to function

- // Primitive types(int,char,double etc.) are passed by value.
- // Objects are **passed by reference.**

```
class Test {
int a;
Test(int i)
{
a = i;
}
void calc(Test o)
{
o.a *= 2;
}
void calc(int a)
{
a*=2;
}
}
```

OUTPUT
Object parameter
Before call: 15
After call: 30
Integer parameter
Before call: 15
After call: 15

```
class Obcall {
public static void main(String args[])
{
Test ob = new Test(15);
System.out.println("Object parameter");
System.out.println("Before call: " + ob.a );
ob.calc(ob);    // //Call by reference
System.out.println("After call: " + ob.a );

int a=15;
System.out.println("Integer parameter");
System.out.println("Before call: " + a);
ob.calc(a);      //Call by value
System.out.println("After call: " + a);  } }
```

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.