Programming Language II
CSE-215

Prof. Dr. Mohammad Abu Yousuf

yousuf@juniv.edu

Comparison between C++ and Java

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications.
Goto	C++ supports goto statement.	Java doesn't support goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write pointer program in C++.	Java supports pointer internally. But you can't write the pointer program in java. It means java has restricted pointer support in java.

Compiler and Interpreter	C++ uses compiler only.	Java uses compiler and interpreter both.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comment.	Java supports documentation comment (/** ... */) to create documentation for java source code.

Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
unsigned right shift >>>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.
Inheritance Tree	C++ creates a new inheritance tree always.	Java uses single inheritance tree always because all classes are the child of Object class in java. Object class is the root of inheritance tree in java.

Variable :

- There are three types of variables: local, instance and static.
- **Local Variable:**
- A variable that is declared inside the method is called local variable.
- **Instance Variable:**
- A variable that is declared inside the class but outside the method is called instance variable . It is not declared as static.
- **Static variable:**
- A variable that is declared as static is called static variable. It cannot be local.

- Example:

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

Control Statements

Java's Selection Statements

- Java supports two selection statements: **if and switch.**

- **These statements allow you to** control the flow of your program's execution based upon conditions known only during run time.
- There are various types of if statement in java.
  - if statement
  - if-else statement
  - nested if statement
  - if-else-if ladder

- **Java If statement:**
  - if (*condition) statement1;*

```
public class IfExample {

public static void main(String[] args) {

    int age=20;

    if(age>18){

        System.out.print("Age is greater than 18");

    }

}

}
```

- Java IF-else Statement
  - if (*condition) statement1;*

    else *statement2;*

```
public class IfElseExample {

public static void main(String[] args) {

    int number=13;

    if(number%2==0){

        System.out.println("even number");

    }else{

        System.out.println("odd number");

    }

}

}
```

- Java IF-else-if ladder Statement

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

- Java Nested ifs:

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d; // this if is
  else a = c;        // associated with this else
}
else a = d;          // this else refers to if(i == 10)
```

- Java Switch Statement:

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......

default:
 code to be executed if all cases are not matched;
}
```

expression must be of type ***byte, short, int, char,*** or an enumeration.
Expression can also be of type ***String.***
Each value specified in the case statements must be a unique constant expression (such as a literal value). Duplicate **case values are not** allowed. The type of each value must be compatible with the type of *expression.*

- Example:

```
public class SwitchExample {
public static void main(String[] args) {
    int number=20;
    switch(number){
    case 10: System.out.println("10");break;
    case 20: System.out.println("20");break;
    case 30: System.out.println("30");break;
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

- The break statement is optional. If you omit the break, execution will continue on into the next case.

- It means it executes all statement after first match if break statement is not used with switch cases.

- It is sometimes desirable to have multiple cases without break statements between them.

```
public class SwitchExample2 {
public static void main(String[] args) {
    int number=20;
    switch(number){
    case 10: System.out.println("10");
    case 20: System.out.println("20");
    case 30: System.out.println("30");
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

Output:

20
30
Not in 10, 20 or 30

- For example:

```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String args[]) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
          break;
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i is less than 10");
          break;
        default:
          System.out.println("i is 10 or more");
      }
  }
}
```

- Output:

```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

- **Nested switch Statements:**

```
switch(count) {
  case 1:
    switch(target) { // nested switch
      case 0:
        System.out.println("target is zero");
        break;
      case 1: // no conflicts with outer switch
        System.out.println("target is one");
        break;
    }
    break;
  case 2: // …
```

Iteration Statements

- Java's iteration statements are **for, while, and do-while. These statements create what** we commonly call *loops.*

For loop:

```
for(initialization;condition;incr/decr){
//code to be executed
}
```

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

- **Using the Comma in for loop:**

```
class Sample {
  public static void main(String args[]) {
    int a, b;

    b = 4;
    for(a=1; a<b; a++) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      b--;
    }
  }
}
```

```
// Using the comma.
class Comma {
  public static void main(String args[]) {
    int a, b;

    for(a=1, b=4; a<b; a++, b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
    }
  }
}
```

- **Some for Loop Variations**

```
boolean done = false;

for(int i=1; !done; i++) {
  // …
  if(interrupted()) done = true;
}
```

In this example, the **for loop continues to run until the boolean variable done is set to true. It does not test the value of i.**

```
// Parts of the for loop can be empty.
class ForVar {
  public static void main(String args[]) {
    int i;
    boolean done = false;

    i = 0;
    for( ; !done; ) {
      System.out.println("i is " + i);
      if(i == 10) done = true;
      i++;
    }
  }
}
```

Here, the initialization and iteration expressions have been moved out of the **for. Thus,** parts of the **for are empty.**

Java For-each Loop:

- The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

- It works on elements basis not index. It returns element one by one in the defined variable.

- **Java For-each Loop:**

```
for(Type var:array){
//code to be executed
}
```

```
public class ForEachExample {
public static void main(String[] args) {
    int arr[]={12,23,44,56,78};
    for(int i:arr){
        System.out.println(i);
    }
}
}
```

Output:
12
23
44
56
78

Java Labeled For Loop:

- We can have name of each for loop. To do so, we use label before the for loop. It is useful if we have nested for loop so that we can break/continue specific for loop.
- Normally, break and continue keywords breaks/continues the inner most for loop only.

```
labelname:
for(initialization;condition;incr/decr){
//code to be executed
}
```

```
public class LabeledForExample {
public static void main(String[] args) {
    aa:
        for(int i=1;i<=3;i++){
            bb:
                for(int j=1;j<=3;j++){
                    if(i==2&&j==2){
                        break aa;
                    }
                    System.out.println(i+" "+j);
                }
        }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
```

If you use **break bb;**, it will break inner loop only which is the default behavior of any loop.

Using break to Exit a Loop:

- By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

- Example:

```
// Using break to exit a loop.
class BreakLoop {
  public static void main(String args[]) {
    for(int i=0; i<100; i++) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
  }
}
```

- When used inside a set of nested loops, the break statement will only break out of the innermost loop.

```
// Using break with nested loops.
class BreakLoop3 {
  public static void main(String args[]) {
    for(int i=0; i< 3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break; // terminate loop if j is 10
        System.out.print(j + " ");
      }
      System.out.println();
    }
    System.out.println("Loops complete.");
  }
}
```

```
Pass 0: 0 1 2 3 4 5 6 7 8 9
Pass 1: 0 1 2 3 4 5 6 7 8 9
Pass 2: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

- **Using continue:**
- Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.
- The continue statement performs such an action.

```
// Demonstrate continue.
class Continue {
  public static void main(String args[]) {

    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;
      System.out.println("");
    }
  }
}
```

- As with the break statement, continue may specify a label to describe which enclosing loop to continue.

```
// Using continue with a label.
class ContinueLabel {
  public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
         for(int j=0; j<10; j++) {
           if(j > i) {
             System.out.println();
             continue outer;
           }
           System.out.print(" " + (i * j));
         }
       }
       System.out.println();
  }
}
```

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

return statement:

- The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.
- As you can see, the **final println()** statement is not executed. As soon as return is executed, control passes back to the caller.

```
// Demonstrate return.
class Return {
  public static void main(String args[]) {
    boolean t = true;

    System.out.println("Before the return.");

    if(t) return; // return to caller

    System.out.println("This won't execute.");
  }
}
```

Output:
Before the return.

Thank you