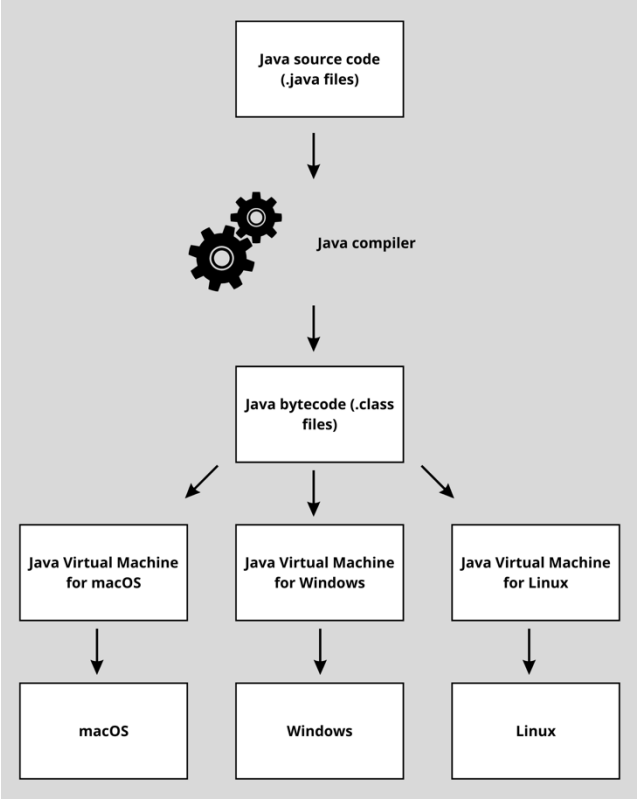
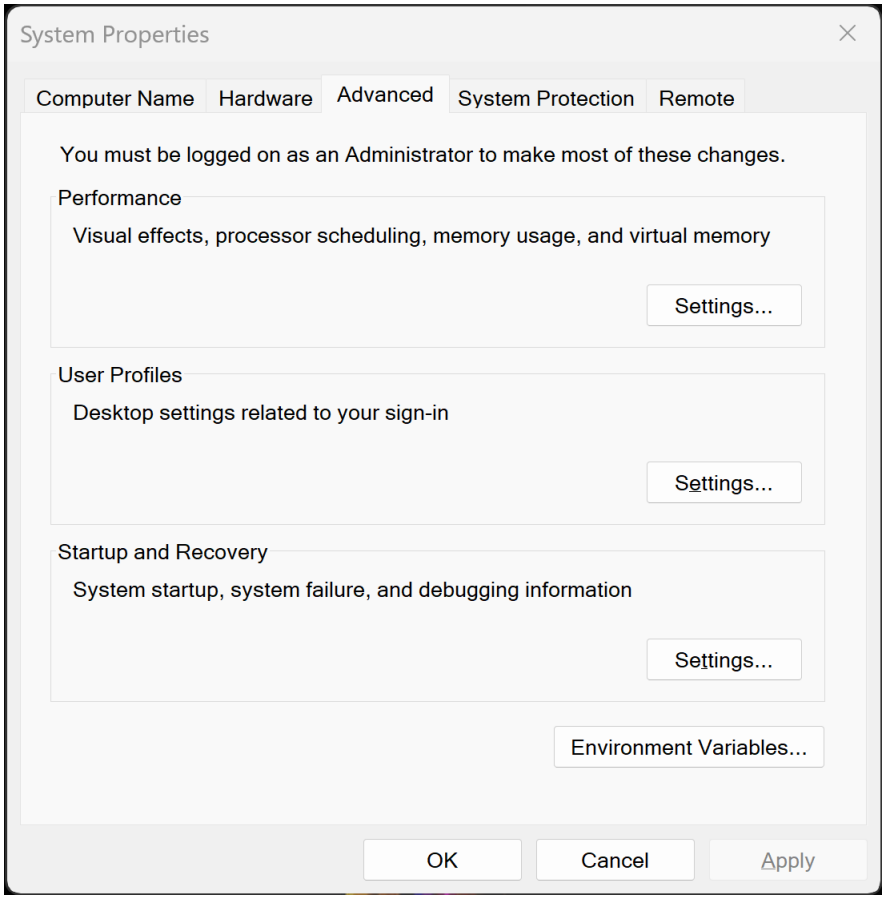
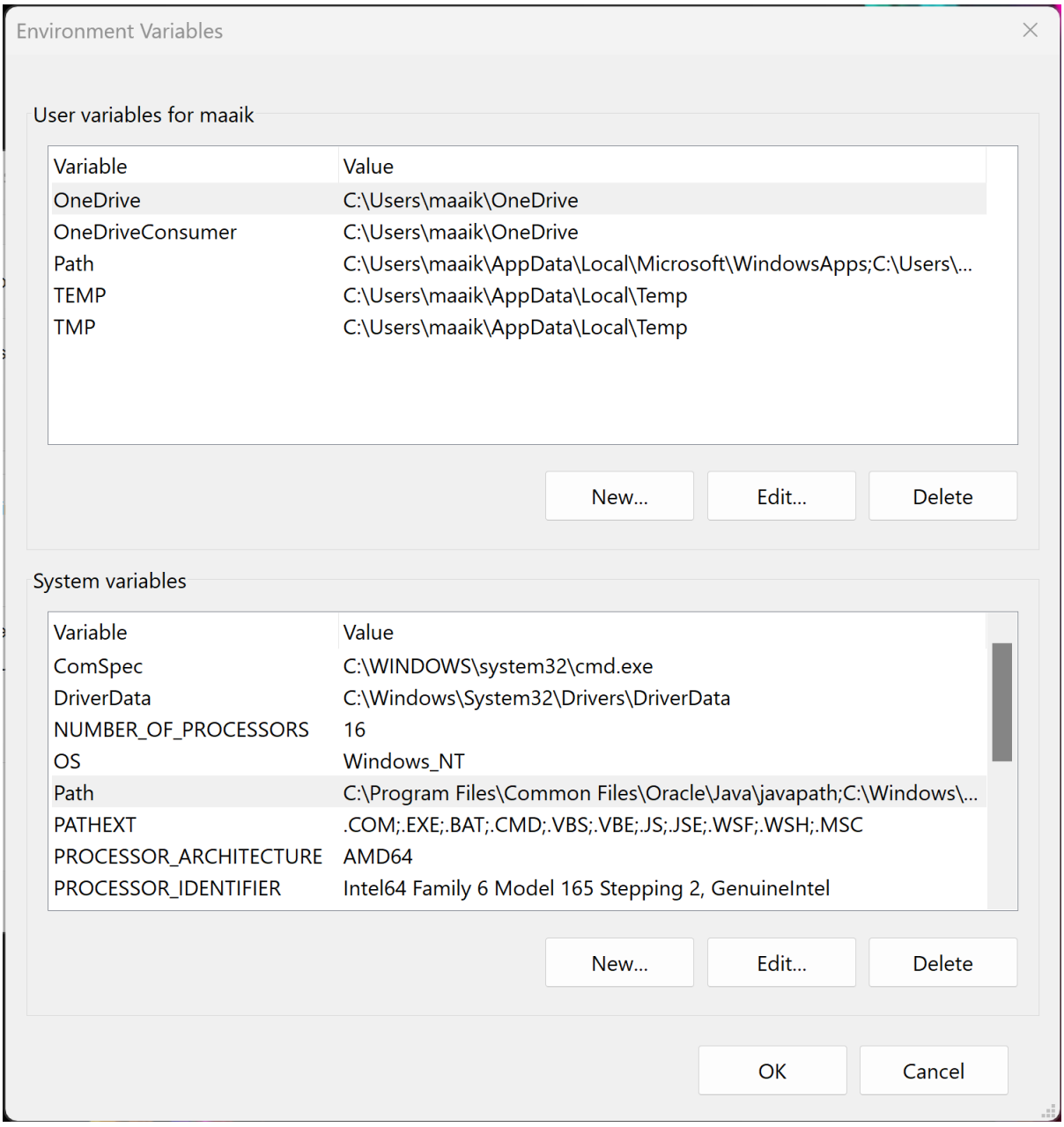
# Chapter 1: Getting Started with Java

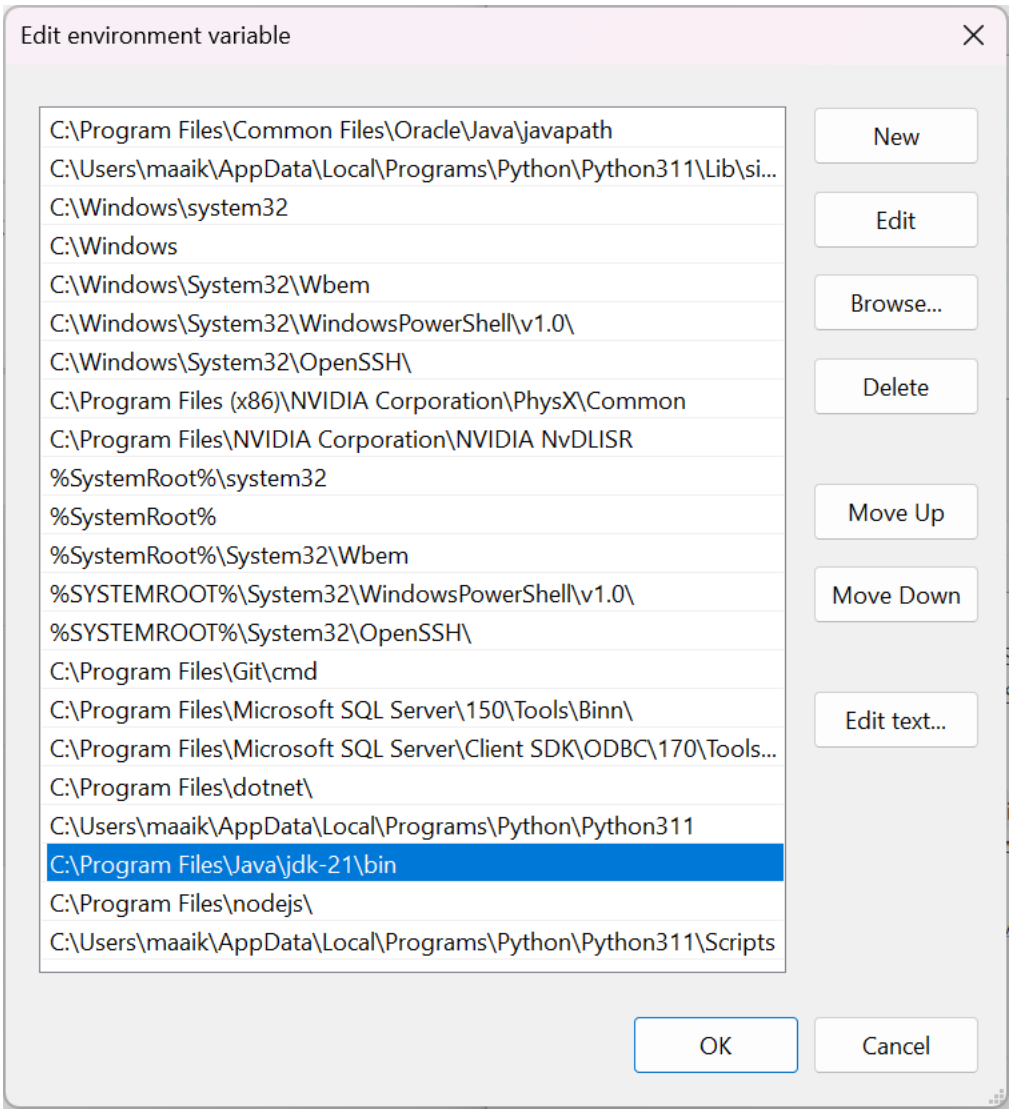


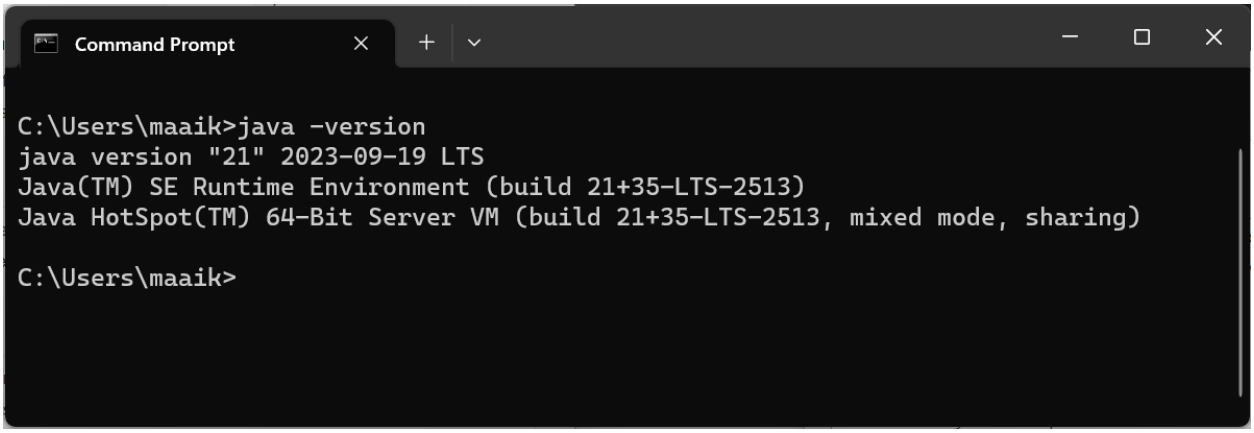

```
maaikevanputten — -bash — 72×11
Last login: Tue Mar 14 11:18:00 on ttys000

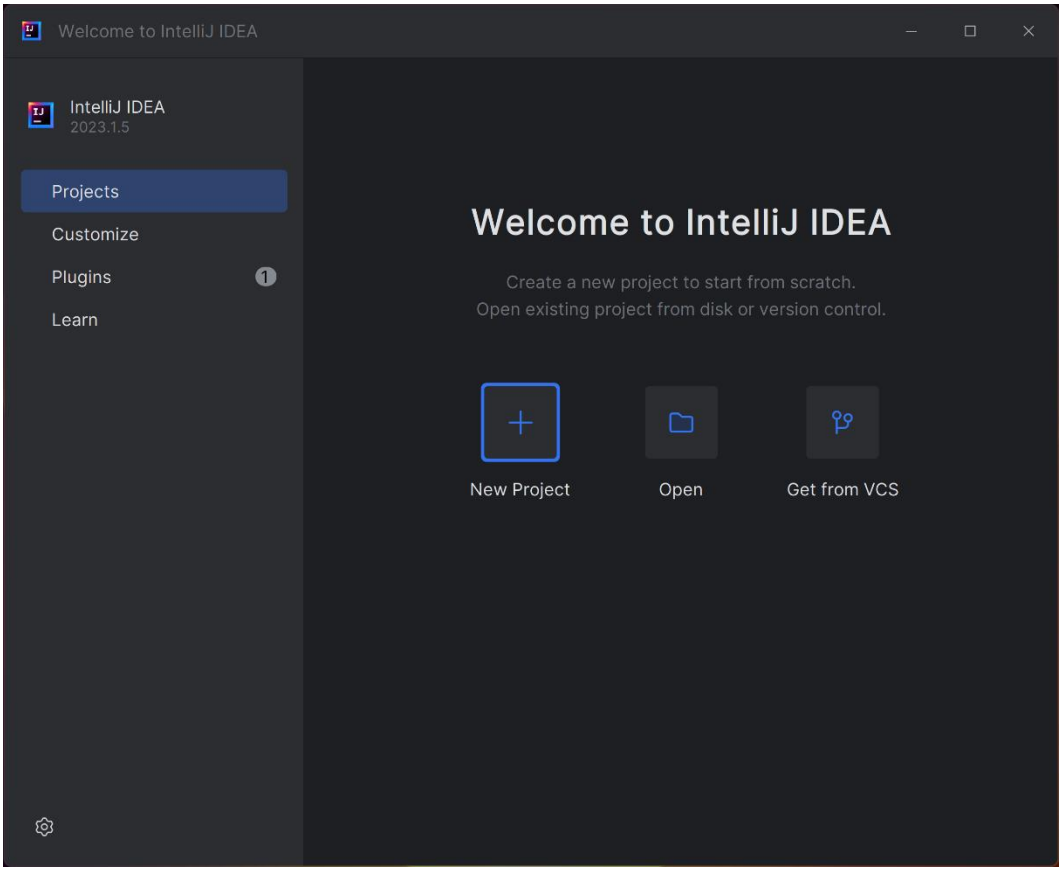
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
Maaikes-MBP:~ maaikevanputten$ java --version
openjdk 19.0.1 2022-10-18
OpenJDK Runtime Environment (build 19.0.1+10-21)
OpenJDK 64-Bit Server VM (build 19.0.1+10-21, mixed mode, sharing)
Maaikes-MBP:~ maaikevanputten$
```

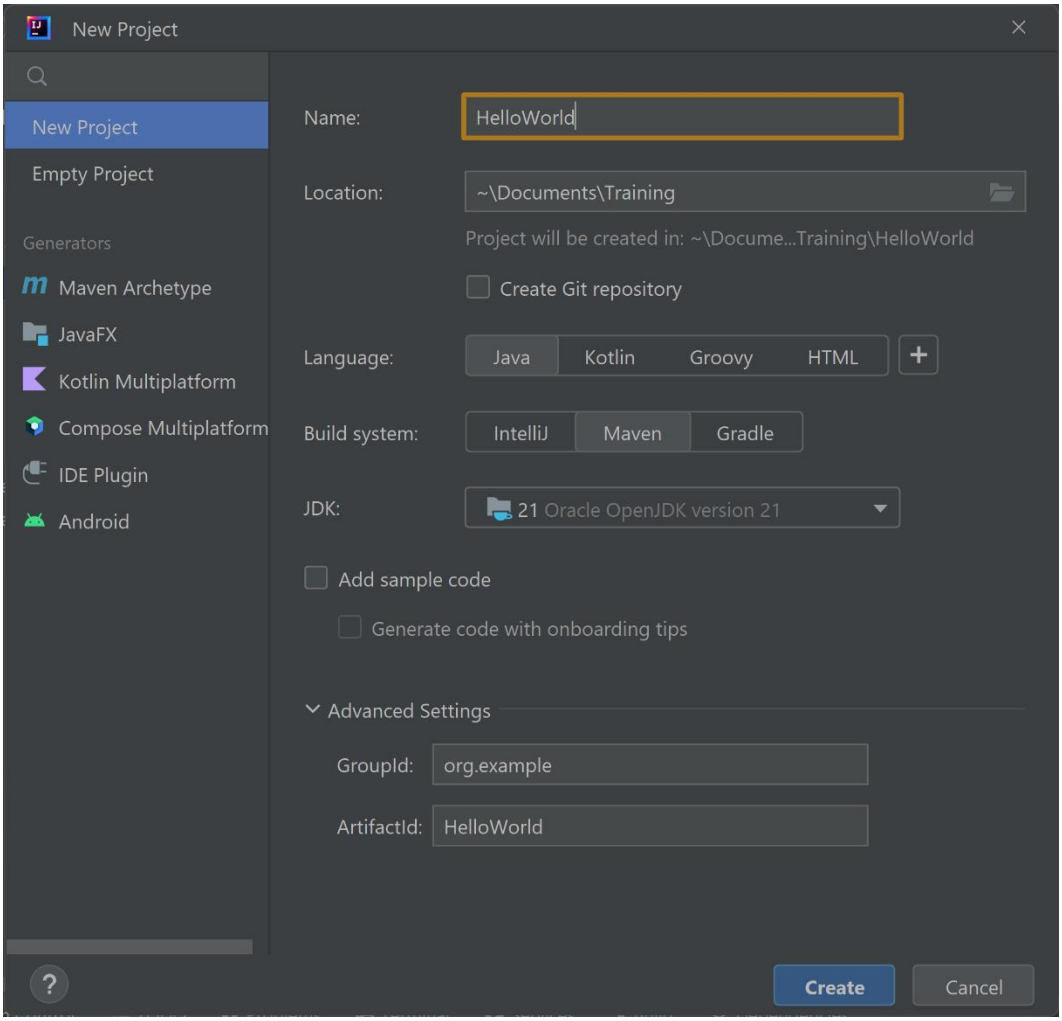

System Properties
Computer Name
Hardware
Advanced
System Protection
Remote
You must be logged on as an Administrator to make most of these changes.
Performance
Visual effects, processor scheduling, memory usage, and virtual memory
Settings...
User Profiles
Desktop settings related to your sign-in
Settings...
Startup and Recovery
System startup, system failure, and debugging information
Settings...
Environment Variables...
OK
Cancel
Apply

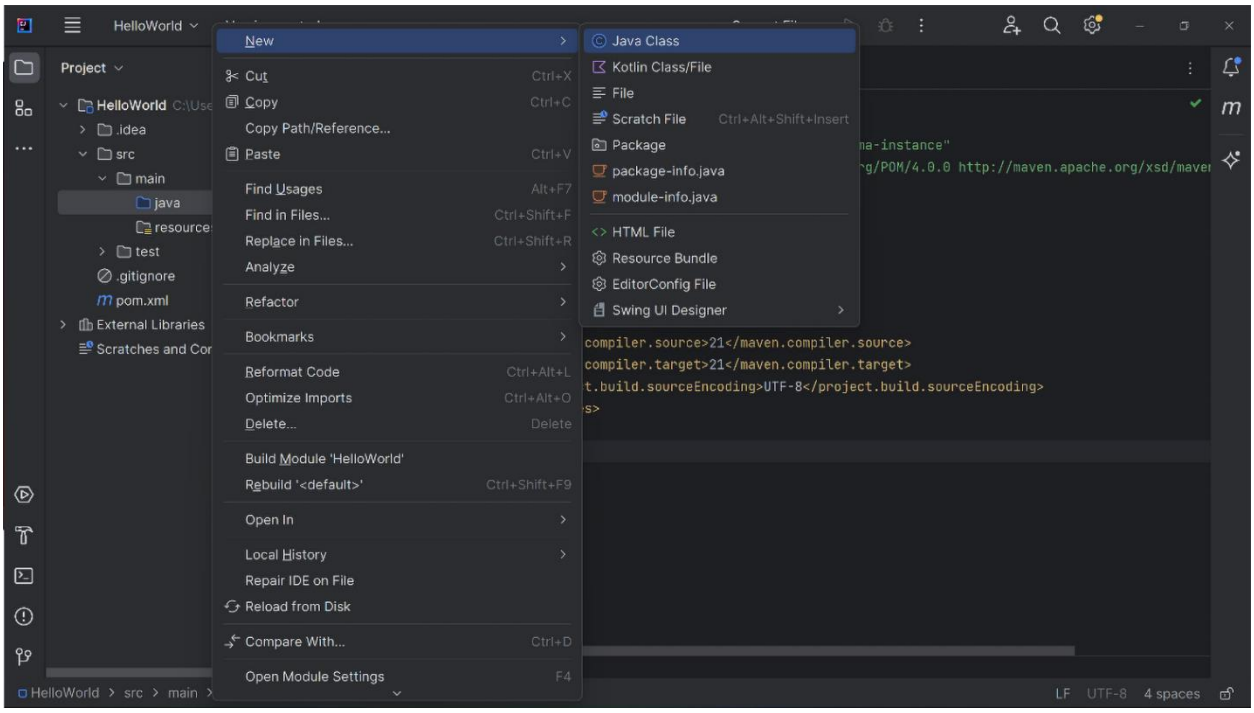

Environment Variables
User variables for maaik
Variable
Value
OneDrive
C:\Users\maaik\OneDrive
OneDriveConsumer
C:\Users\maaik\OneDrive
Path
C:\Users\maaik\AppData\Local\Microsoft\WindowsApps;C:\Users\...
TEMP
C:\Users\maaik\AppData\Local\Temp
TMP
C:\Users\maaik\AppData\Local\Temp
New...
Edit...
Delete
System variables
Variable
Value
ComSpec
C:\WINDOWS\system32\cmd.exe
DriverData
C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS
16
OS
Windows_NT
Path
C:\Program Files\Common Files\Oracle\Java\javapath;C:\Windows\...
PATHEXT
.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE
AMD64
PROCESSOR_IDENTIFIER
Intel64 Family 6 Model 165 Stepping 2, GenuineIntel
New...
Edit...
Delete
OK
Cancel

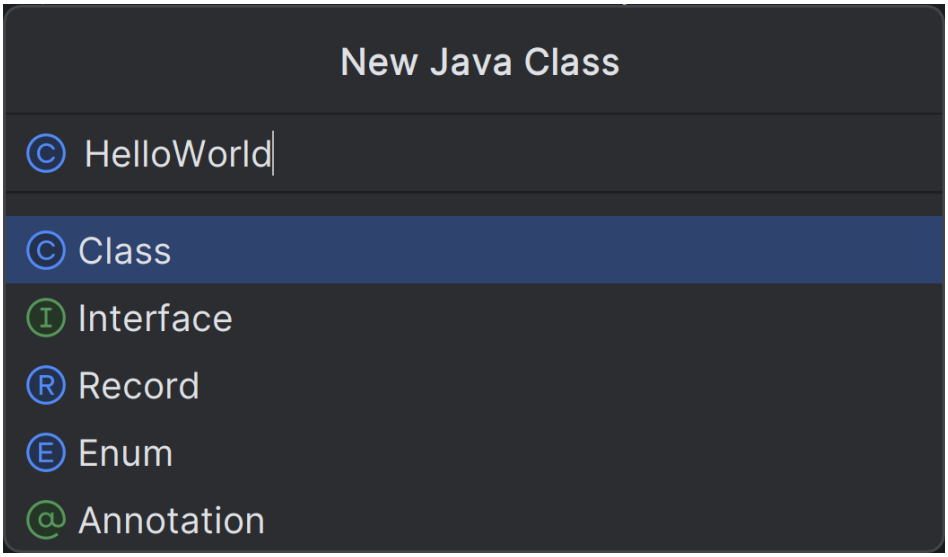

Edit environment variable
C:\Program Files\Common Files\Oracle\Java\javapath
C:\Users\maaik\AppData\Local\Programs\Python\Python311\Lib\si...
C:\Windows\system32
C:\Windows
C:\Windows\System32\Wbem
C:\Windows\System32\WindowsPowerShell\v1.0\
C:\Windows\System32\OpenSSH\
C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common
C:\Program Files\NVIDIA Corporation\NVIDIA NvDLISR
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
C:\Program Files\Git\cmd
C:\Program Files\Microsoft SQL Server\150\Tools\Binn\
C:\Program Files\Microsoft SQL Server\Client SDK\ODBC\170\Tools...
C:\Program Files\dotnet\
C:\Users\maaik\AppData\Local\Programs\Python\Python311
C:\Program Files\Java\jdk-21\bin
C:\Program Files\nodejs\
C:\Users\maaik\AppData\Local\Programs\Python\Python311\Scripts
New
Edit
Browse...
Delete
Move Up
Move Down
Edit text...
OK
Cancel


Command Prompt
C:\Users\maaik>java -version
java version "21" 2023-09-19 LTS
Java(TM) SE Runtime Environment (build 21+35-LTS-2513)
Java HotSpot(TM) 64-Bit Server VM (build 21+35-LTS-2513, mixed mode, sharing)
C:\Users\maaik>


Welcome to IntelliJ IDEA
IntelliJ IDEA
2023.1.5
Projects
Customize
Plugins
1
Learn
Welcome to IntelliJ IDEA
Create a new project to start from scratch.
Open existing project from disk or version control.
New Project
Open
Get from VCS


New Project
New Project
Empty Project
Generators
Maven Archetype
JavaFX
Kotlin Multiplatform
Compose Multiplatform
IDE Plugin
Android
Name:
HelloWorld
Location:
~\Documents\Training
Project will be created in: ~\Docume...Training\HelloWorld
Create Git repository
Language:
Java
Kotlin
Groovy
HTML
Build system:
IntelliJ
Maven
Gradle
JDK:
21 Oracle OpenJDK version 21
Add sample code
Generate code with onboarding tips
Advanced Settings
GroupId:
org.example
ArtifactId:
HelloWorld
Create
Cancel


HelloWorld
Project
HelloWorld
.idea
src
main
java
resources
test
.gitignore
pom.xml
External Libraries
Scratches and Cor
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path/Reference...
Paste Ctrl+V
Find Usages Alt+F7
Find in Files... Ctrl+Shift+F
Replace in Files... Ctrl+Shift+R
Analyze
Refactor
Bookmarks
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Delete... Delete
Build Module 'HelloWorld'
Rebuild '<default>' Ctrl+Shift+F9
Open In
Local History
Repair IDE on File
Reload from Disk
Compare With... Ctrl+D
Open Module Settings F4
Java Class
Kotlin Class/File
File
Scratch File Ctrl+Alt+Shift+Insert
Package
package-info.java
module-info.java
HTML File
Resource Bundle
EditorConfig File
Swing UI Designer
compiler.source>21</maven.compiler.source>
compiler.target>21</maven.compiler.target>
t.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
HelloWorld > src > main
LF UTF-8 4 spaces


New Java Class
HelloWorld
Class
Interface
Record
Enum
Annotation

HelloWorld.java

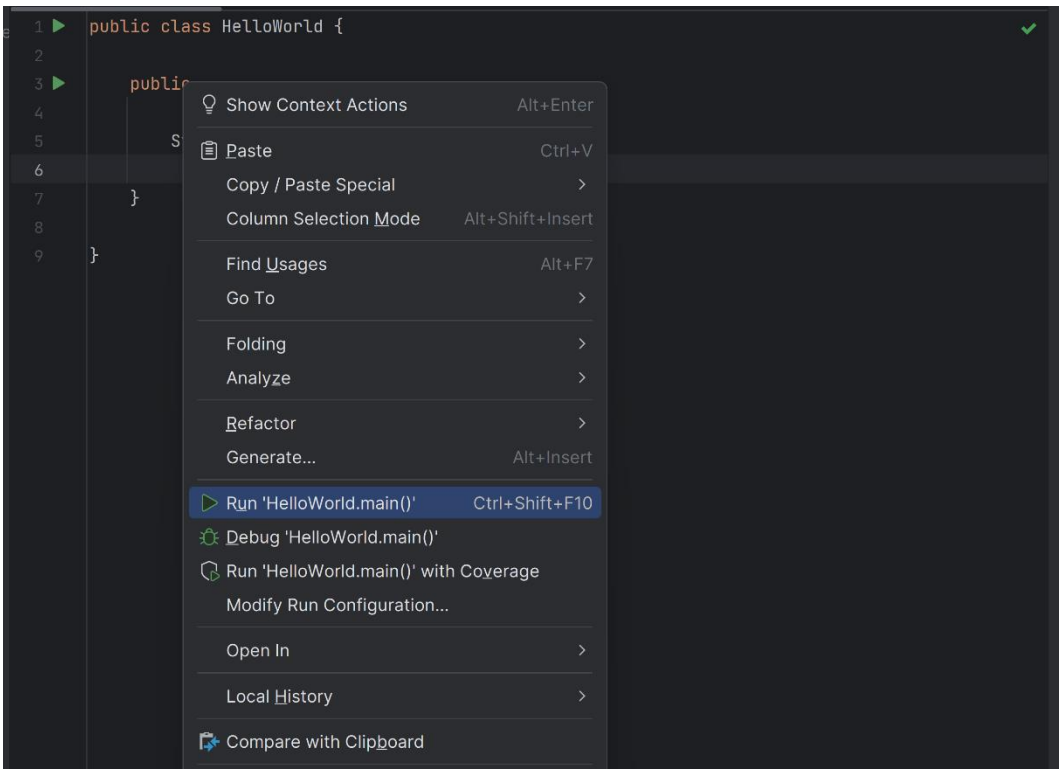
```java
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello world!");

    }

}
```



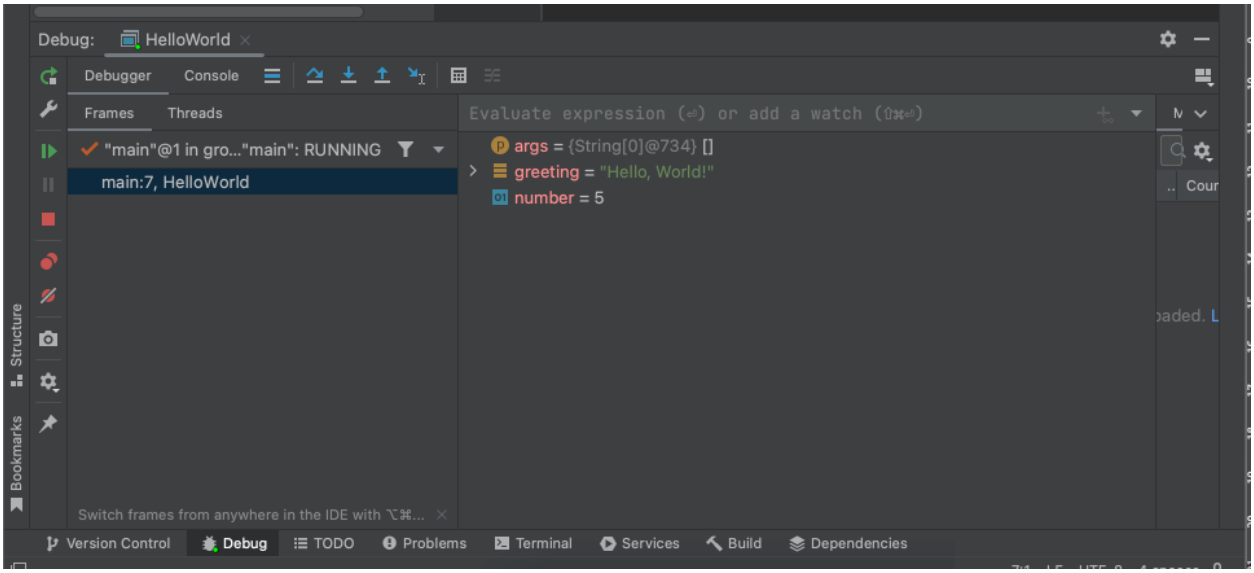

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-
Hello world!

Process finished with exit code 0
```

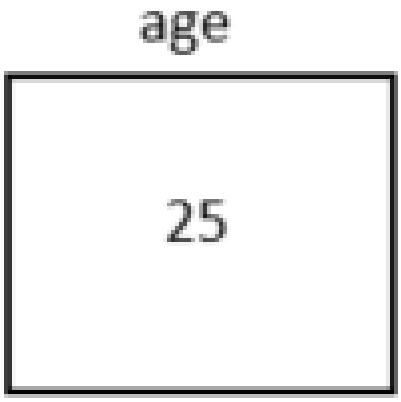
```java
public class HelloWorld {
    no usages
    public static void main(String[] args) {
        String greeting = "Hello, World!";
        System.out.println(greeting);

        int number = 5;
        int doubled = doubleNumber(number);
        System.out.println("The doubled number is: " + doubled);
    }

    1 usage
    public static int doubleNumber(int input) {
        return input * 2;
    }
}
```



## Chapter 2: Variables and Primitive Data Types



## Chapter 3: Operators and Casting

```
25          int x=3;
26          ++x;
27          System.out.println(x);  // 4
28          System.out.println(x++);// 4
29          System.out.println(x);  // 5
30
31          int y=4;
32          y--;
33          System.out.println(y);  // 3
34          System.out.println(--y);// 2
35          System.out.println(y);  // 2
```

```
37      int x = +6;
38      int y = -x;
39      System.out.println(x);  // 6
40      System.out.println(y);  // -6
41
42      int z = (int)3.45;
43      System.out.println(z);  // 3
44
45      boolean b = true;
46      System.out.println(!b); // false
47      System.out.println(b);  // true
```

```java
int res = 3 + 2 * 4;
System.out.println(res); // 11
res = (3 + 2) * 4;
System.out.println(res); // 20
res = 6 + 4 - 2;
System.out.println(res); // 8
res = 10 / 4 * 6 % 10;
System.out.println(res); // 2
```

```java
int a=3, b=2;
int res = a + b;
System.out.println(res); // 5
String s = "abc";
String s1 = a + s;  // "3abc"
String s2 = s + a;  // "abc3"
System.out.println(s1 + " " + s2); // "3abc abc3"

System.out.println("Output is "+ a + b); // "Output is 32"
System.out.println("Output is "+ (a + b)); // "Output is 5"
```

```java
int x=3, y=4;
System.out.println(x == y); // false
System.out.println(x != y); // true
System.out.println(x > y);  // false
System.out.println(x >= y); // false
System.out.println(x < y);  // true
System.out.println(x <= y); // true
```

```java
char c   = 'a'; // normal
int i    = c;   // widening, char to int
float f  = i;   // widening, int to float
double d = f;   // widening, float to double
float f2 = 1L;  // widening, long to float
```

```java
// Narrowing
int i   = (int)3.3;                     // narrowing, double to int
byte b  = (byte) 233;                   // narrowing, int to byte
float f = (float) 3.5;                  // narrowing, double to float
System.out.println(i + " " + b + " " +f);   // 3 -23 3.5
```

```java
char c = 12;
char c2 = 90_000;
short s = 12;
s = c;
s = (short) c;
c = s;
c = (char) s;

final char c1 = 12;
short s1 = 12;
s1 = c1;
```

# Chapter 4: Conditional Statements

```
3  public class Scope {
4      public static void main(String[] args) {
5          int x = 1;
6          x++;
7
8          { // start of block
9              int y = 2;
10             y++;
11             x++;
12         } // end of block
13         x++;
14         y++; // out of scope
15     }
16 }
```

if (booleanExpression) statement/block

[ else if (booleanExpression) statement/block ] ...

[ else statement/block ]

```
8          int x=5, y=4;
9          if(x > y)
10             System.out.println(x + " > "+y);
11         if(x < y)
12             System.out.println(x + " < "+y);
13         if(x == y){
14             String s = x + " == "+y;
15             System.out.println(s);
16         }
```

```java
int x=4, y=5;
if(x > y) {
    System.out.println(x + " > " + y);
} else if(x < y) {
    System.out.println(x + " < " + y);
} else if(x == y){
    System.out.println(x + " == " + y);
}
System.out.println("Here");
```

```java
int x=4, y=4;
if(x > y) {
    System.out.println(x + " > " + y);
} else if(x < y) {
    System.out.println(x + " < " + y);
} else {
    System.out.println(x + " == " + y);
}
System.out.println("Here");
```

```java
final int JAN = 1;  final int FEB = 2;   final int MAR = 3; // constants
final int APR = 4;  final int MAY = 5;   final int JUN = 6;
final int JUL = 7;  final int AUG = 8;   final int SEP = 9;
final int OCT = 10; final int NOV = 11;  final int DEC = 12;

Scanner sc = new Scanner(System.in);    // import java.util.Scanner;
System.out.print("Enter month --> ");
int month = sc.nextInt();
```

```java
int numDays=0;
if(month == JAN || month == MAR || month == MAY || month == JUL
        || month == AUG || month == OCT || month == DEC) {
    numDays=31;
} else if (month == APR || month == JUN || month == SEP || month == NOV) {
    numDays=30;
} else if (month == FEB) {
    System.out.print("Enter year --> ");
    int year = sc.nextInt();

    if( (year % 400 == 0) || (year % 4 == 0 && !(year % 100 == 0)) ){
        numDays = 29; // leap year e.g. 2000, 2012, 2016
    }else{
        numDays = 28;   // 1900 (divisible by 100)
    }
} else {
    System.out.println("Invalid month: "+month);
}
if(numDays > 0){
    System.out.println("Number of days is: "+numDays);
}
```

```
switch (expression){
      [case label:
          statement(s);
          break;] ...
       [default:
          statement(s);
          break;]
}
```

```java
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a sport --> ");
        String sport = sc.next();
        switch(sport){
            case "Soccer":
                System.out.println("I play soccer");
                break;
            case "Rugby":
                System.out.println("I play Rugby");
                break;
            default:// can be moved around
                System.out.println("Unknown sport");
                break;
        }
```

```java
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter a number (1..10) --> ");
    int number = sc.nextInt();
    final int two = 2; // compile-time constant
    switch(number){
        case 1:
        case 3:
        case 5:
        case 7:
        case 9:
            System.out.println(number + " is odd.");
            break;
        case two:
        case 4: case 6: case 8: case 10:
            System.out.println(number + " is even.");
            break;
        default:
            System.out.println( number + " is outside range (1..10).");
            break;
    }
```

```
22          int numDays=0;
23          switch(month){
24              case JAN:case MAR:case MAY:case JUL: case AUG:case OCT:case DEC:
25                  numDays=31;
26                  break;
27              case APR:case JUN:case SEP:case NOV:
28                  numDays=30;
29                  break;
30              case FEB:
31                  System.out.print("Enter year --> ");
32                  int year = sc.nextInt();
33  //                if( (A)         || (        B         &&       C)    ){
34                  if( (year % 400 == 0) || (year % 4 == 0 && !(year % 100 == 0)) ){
35                      numDays = 29; // leap year e.g. 2000, 2012, 2016
36                  }else{
37                      numDays = 28;   // 1900 (divisible by 100)
38                  }
39                  break;
40              default:
41                  System.out.println("Invalid month: "+month);
42                  break;
43
44          }
45          if(numDays > 0){
46              System.out.println("Number of days is: "+numDays);
47          }
```

```
58          byte b = 3;
59          switch(b){
60              case 127: case -128:
61                  System.out.println("ok");
62                  break;
63              case 128:
64              case 12:
65              case 12:
66          }
```

```java
int nLetters=0;
String name="Jane";
switch(name){
    case "Jane":
    case "Sean":
    case "Alan":
    case "Paul":
        nLetters = 4;
        break;
    case "Janet":
    case "Susan":
        nLetters = 5;
        break;
    case "Maaike":
    case "Alison":
    case "Miriam":
        nLetters = 6;
        break;
    default:
        System.out.println("Unrecognized name: "+name);
        nLetters = -1;
        break;
}
System.out.println(nLetters);
```

```java
nLetters = switch(name){
    case "Jane", "Sean", "Alan", "Paul" -> 4;
    case "Janet", "Susan" -> 5;
    case "Maaike", "Alison" , "Miriam" -> 6;
    default -> {
        System.out.println("Unrecognized name: "+name);
        yield -1; // 'nLetters' initialized to -1
    }
};
System.out.println(nLetters);
```

```
53          System.out.println(switch(name){
54              case "Jane", "Sean", "Alan", "Paul" -> 4;
55              case "Janet", "Susan" -> 5;
56              case "Maaike", "Alison" , "Miriam" -> 6;
57              default -> "Unrecognized name: "+name;
58          });
```

```
60          nLetters = switch(name){
61              case "Jane", "Sean", "Alan", "Paul" -> {
62                  System.out.println("There are 4 letters in: " + name);
63                  yield 4;
64              }
65              case "Janet", "Susan" -> {
66                  System.out.println("There are 5 letters in: "+name);
67                  yield 5;
68              }
69              case "Maaike", "Alison" , "Miriam" -> {
70                  System.out.println("There are 6 letters in: "+name);
71                  yield 6;
72              }
73              default -> {
74                  System.out.println("Unrecognized name: "+name);
75                  yield -1;
76              }
77          };
78          System.out.println(nLetters);
```

```
82          switch(name){
83              case "Jane", "Sean", "Alan", "Paul" -> nLetters = 4;
84              case "Janet", "Susan" -> nLetters = 5;
85              case "Maaike", "Alison" , "Miriam" -> nLetters = 6;
86              default -> {
87                  System.out.println("Unrecognized name: "+name);
88                  nLetters = -1;
89              }
90          }
91          System.out.println(nLetters);
```

```
94          nLetters = switch(name){
95              case "Jane":
96              case "Sean":
97              case "Alan":
98              case "Paul":
99                  System.out.println("There are 4 letters in: " + name);
100                 yield 4;
101             case "Janet":
102             case "Susan":
103                 System.out.println("There are 5 letters in: "+name);
104                 yield 5;
105             case "Maaike":
106             case "Alison":
107             case "Miriam":
108                 System.out.println("There are 6 letters in: "+name);
109                 yield 6;
110             default:
111                 System.out.println("Unrecognized name: "+name);
112                 yield -1;
113         };
114         System.out.println(nLetters);
```

# Chapter 5: Understanding Iteration

```
while (booleanExprIsTrue) {

  // do something

}
```

```java
int x=1;

while (x <= 3){
    System.out.println("Loop: "+x); // 1, 2, 3
    x++;
}
System.out.println("Final x value is: "+x); // 4
```

```java
int sum = 0;
boolean keepGoing=true;
while(keepGoing){
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter a number (negative number to exit) --> ");
    int n = sc.nextInt();
    if(n < 0){
        keepGoing = false;
    } else{
        sum = sum + n; // sum += n
    }
}
System.out.println("Sum of numbers is: "+sum);
```

```
do {

  // do something

} while (booleanExprIsTrue);
```

```
19      int sum = 0;
20      boolean keepGoing=true;
21      do {
22          Scanner sc = new Scanner(System.in);
23          System.out.println("Enter a number (negative number to exit) --> ");
24          int n = sc.nextInt();
25          if(n < 0){
26              keepGoing = false;
27          } else{
28              sum = sum + n; // sum += n
29          }
30      } while(keepGoing);
31      System.out.println("Sum of numbers is: "+sum);
```

```
49      Scanner sc = new Scanner(System.in);
50      System.out.println("Please enter your age -- > ");
51      int age = sc.nextInt();
52      while(age >= 18){
53          // purchase alcohol...
54          System.out.println("As you are "+age+" years of age, " +
55                  "you can purchase alcohol.");
56
57          System.out.println("Please enter your age -- > ");
58          age = sc.nextInt();
59      }
```

```
49      Scanner sc = new Scanner(System.in);
50      System.out.println("Please enter your age -- > ");
51      int age = sc.nextInt();
52      do{
53          // purchase alcohol...
54          System.out.println("As you are "+age+" years of age, " +
55                  "you can purchase alcohol.");
56
57          System.out.println("Please enter your age -- > ");
58          age = sc.nextInt();
59      }while (age >= 18);
```

```
for (initialization; booleanExpression; incr/decr) {

   // do something

}
```

```java
// for(init; booleanExpr; incr/decr)
for(int i=1; i<=3; i++){
    System.out.println(i); // 1,2,3
}

for(int i=3; i>=1; i--){
    System.out.println(i); // 3,2,1
}
```

```java
for(int i=1; i<=3; i++);{
    System.out.println("Looping"); // only appears once!
}

for(int i=10; i<=50; i+=10){
    System.out.println(i); // 10, 20, 30, 40, 50
}
System.out.println(i); // i is out of scope

for(int i=0, j=0; i<1 && j<1; i++, j++){
    System.out.println(i + " " + j); // 0 0
}
```

```java
int[] ia = {1,2,3};
for(int i=0; i<ia.length; i++){
    System.out.println(ia[i]); // 1, 2, 3
}
```

```
for (dataType variableName : array or collection){

  // do something

}
```

```java
int[] ia = {1,2,3};
for(int n:ia){
    System.out.println(n); // 1, 2, 3
}
```

```java
int[] data = {9, 3, 5, 7};

System.out.println("[]\t[n]\tHistogram");
for (int i=0; i<data.length; i++){
    System.out.print(i + "\t" + data[i] + "\t");
    for(int j=1; j<=data[i]; j++) { // write out data[i] stars
        System.out.print("*");      // print() not println()
    }
    System.out.println();           // go onto next line
}
```

```
[]  [n] Histogram
0   9   *********
1   3   ***
2   5   *****
3   7   *******
```

```
13          System.out.println("i, j");
14          for (int i = 1; i <= 3; i++) {
15              for (int j = 1; j <= 5; j++) {
16                  if (j == 3) {
17                      break; // breaks out of inner loop
18                  }
19                  System.out.println(i + ", " + j);
20              }
21          }
```

```
25      System.out.println("i, j"); // placed BEFORE label!!
26      OUTERLOOP:
27      for (int i = 1; i <= 3; i++) {
28          for (int j = 1; j <= 5; j++) {
29              if (j == 3) {
30                  break OUTERLOOP;// case sensitive
31              }
32              System.out.println(i + ", " + j);
33          }
34      }
35      System.out.println("here");
```

```
32          System.out.println("i, j");
33          for (int i = 1; i <= 3; i++) {
34              for (int j = 1; j <= 5; j++) {
35                  if (j == 3) {
36                      continue; // next iteration
37                  }
38                  System.out.println(i + ", " + j);
39              }
40          }
```

```
28          System.out.println("i, j\n====="); // placed BEFORE label!!
29          OUTERLOOP:
30          for (int i = 1; i <= 3; i++) {
31              for (int j = 1; j <= 5; j++) {
32                  if (j == 3) {
33                      continue OUTERLOOP;// continues with OUTERLOOP
34                  }
35                  System.out.println(i + ", " + j);
36              }
37          }
```

# Chapter 6: Working with Arrays

| Index: | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| | 31 | 7 | 5 | 1 | 0 |

| Index: | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 1 | 5 | 6 | 7 | 8 |
| 2 | 9 | 10 | 11 | 12 |

# Chapter 7: Methods

```java
public class Methods {
    public static void main(String[] args) {
        System.out.println("main: before call to simpleExample()");
        simpleExample(); // method call
        System.out.println("main: after call to simpleExample()");
    }
    public static void simpleExample(){ // method definition
        System.out.println("\tExecuting simpleExample() method...");
    }
}
```

```java
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter a number (1..10) --> ");
        int number = sc.nextInt();
        if(number < 1 || number > 10){
            System.out.println("Invalid number! "+number);
        }

        System.out.print("Enter a number (1..10) --> ");
        number = sc.nextInt();
        if(number < 1 || number > 10){
            System.out.println("Invalid number! "+number);
        }

        System.out.print("Enter a number (1..10) --> ");
        number = sc.nextInt();
        if(number < 1 || number > 10){
            System.out.println("Invalid number! "+number);
        }
```

```java
public class Methods {
    public static void main(String[] args) {
        int number = getNumber();
        number = getNumber();
        number = getNumber();
    }
    public static int getNumber(){
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number (1..10) --> ");
        int number = sc.nextInt();
        if(number < 1 || number > 10){
            System.out.println("Invalid number! "+number);
        }
        return number;
    }
}
```

[access-modifier] [static] **return-type methodName([parameters])** [throws someException] {

// method code

}

```java
// method definition/declaration
public static int performCalc(int x, int y, String operation){  // "parameters"
    int result = switch(operation){
        case "+" -> x + y;
        case "-" -> x - y;
        case "*" -> x * y;
        case "/" -> x / y;
        case "%" -> x % y;
        default -> {
            System.out.println("Unrecognized operation: "+operation);
            yield -1; // error
        }
    };
    return result;
}
```

```java
public static void main(String[] args) {
    int result = performCalc( x: 10,  y: 2,  operation: "+"); // method call; passing down "arguments"
    System.out.println(result); // 12
    System.out.println(performCalc( x: 10,  y: 2,  operation: "-")); // 8
    System.out.println(performCalc( x: 10,  y: 2,  operation: "*")); // 20
    System.out.println(performCalc( x: 10,  y: 2,  operation: "/")); // 5
    performCalc( x: 10,  y: 2,  operation: "%");// return value ignored
    System.out.println(performCalc( x: 10,  y: 2,  operation: "&")); // Unrecognized operation: &, -1
}
public static int performCalc(int x, int y, String operation){  // "parameters"
    int result = switch(operation){
        case "+" -> x + y;
        case "-" -> x - y;
        case "*" -> x * y;
        case "/" -> x / y;
        case "%" -> x % y;
        default -> {
            System.out.println("Unrecognized operation: "+operation);
            yield -1; // error
        }
    };
    return result;
}
```

```java
public static int performCalc(int x, int y, String operation){
```

method signature

```java
public static void someMethod(){}
public static void someMethod(int x){}
public static void someMethod(double x){}
public static void someMethod(String x){}
public static void someMethod(double x, int y){}
public static void someMethod(int x, double y){}
public static void someMethod(int a, double b){}
public static int someMethod(int x, double b){ return 0;}
```

```
6       public static void someMethod(){}
7       public static void someMethod(int x){}
8       public static void someMethod(double x){}
9       public static void someMethod(String x){}
10      public static void someMethod(double x, int y){}
11      public static void someMethod(int x, double y){}
12      public static void someMethod(int a, double b){}
13      public static int someMethod(int x, double b){ return 0;}
```

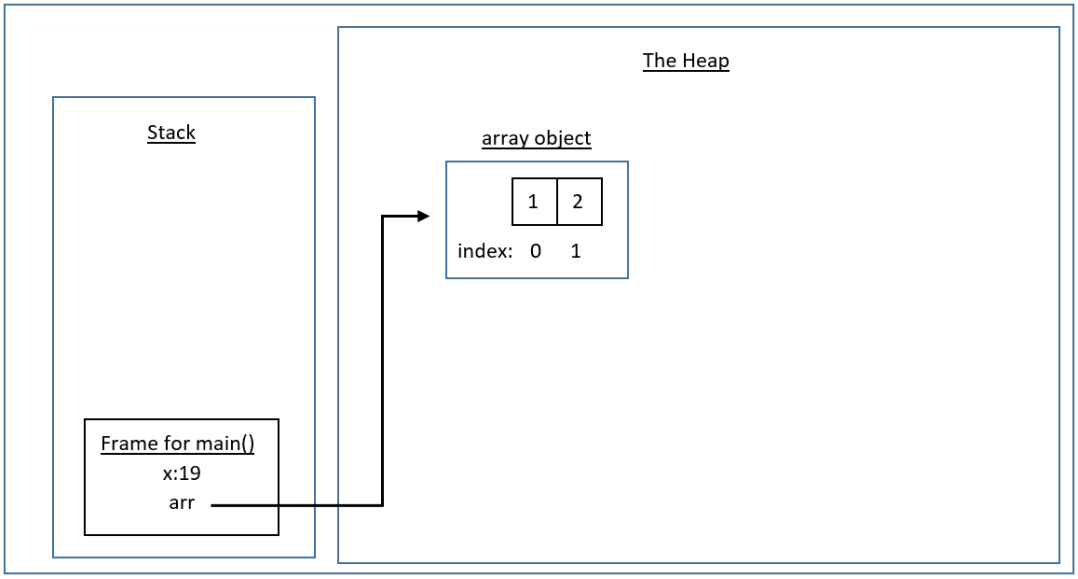
```
3   public class Methods {
4       public static void main(String[] args) {
5           int sum = add( x: 3,  y: 4);
6           System.out.println(sum); // 7
7           double addition = add( x: 3.0,  y: 4.0);
8           System.out.println(addition); // 7.0
9       }
10      public static int add(int x, int y){
11          System.out.println("add(int,int)");
12          return x + y;
13      }
14      public static double add(double x, double y){
15          System.out.println("add(double,double)");
16          return x + y;
17      }
18  }
```

```java
    public static void main(String[] args) {
        m1();                 // 0
        m1( ...args: 1);          // 1
        m1( ...args: 1, 2);       // 3
        m1( ...args: 1, 2, 3);    // 6
    }
    public static void m1(int... args){ // varargs
        int sum = 0;
        for(int i:args){
            sum += i;
        }
        System.out.println(sum);
    }
```

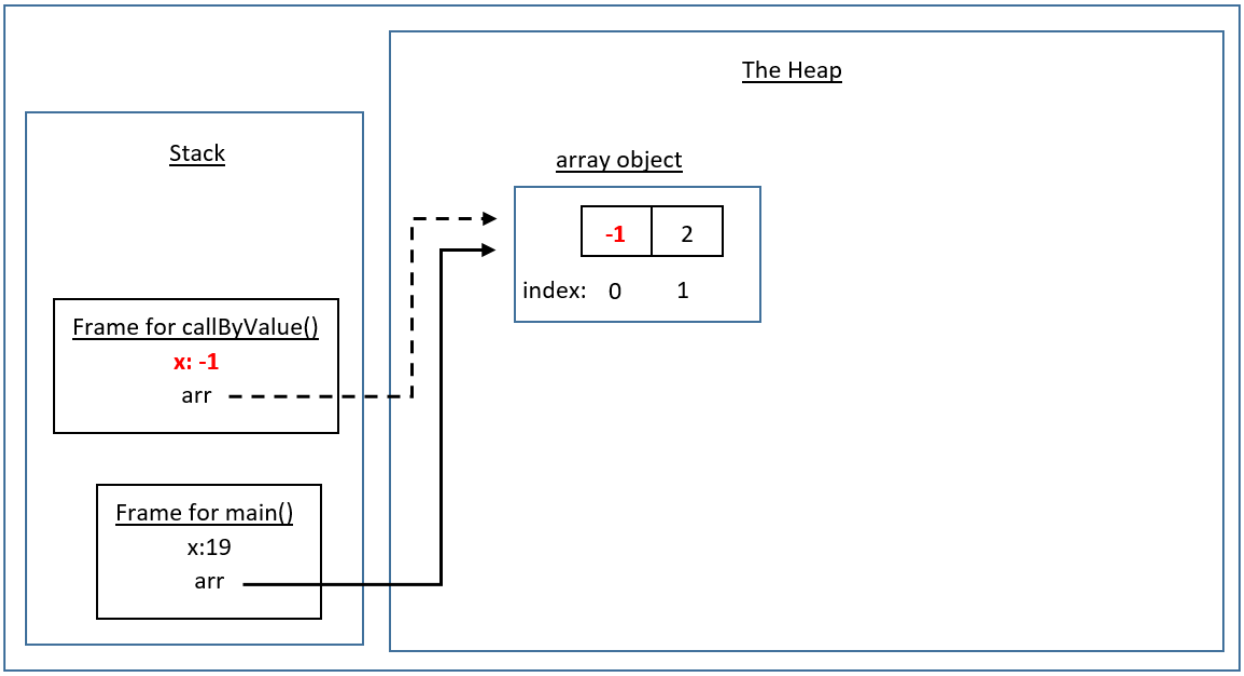
```java
public class Methods {
    public static void main(String[] args) {
        m1();
        m1("A");
        m1("A", "B");
        m1( n: "A",  ...args: "B", "C");
    }
    public static void m1(int n, String... args){}
    public static void m1(String... args, int n){}
    public static void m1(String[] args){} // this is not varargs

//    public static void m1(String... args){ // varargs
//        for(int i=0; i<args.length; i++){
//            System.out.println(args[i]);
//        }
//        for(String s:args){
//            System.out.println(s);
//        }
//    }
}
```

```java
public class Methods {
    public static void main(String[] args) {
        int x = 19;          // primitive
        int[] arr = {1, 2}; // array
    }
}
```




```java
public class Methods {
    public static void main(String[] args) {
        int x = 19;          // primitive
        int[] arr = {1, 2}; // array
        callByValue(x, arr);
        System.out.println(x); // 19, unchanged
        System.out.println(arr[0] + ", " + arr[1]); // -1, 2
        x = callByValue(x, arr);
        System.out.println(x); // -1, changed
    }
    public static int callByValue(int x, int[] arr){
        x       = -1;
        arr[0]  = -1;
        return x;
    }
}
```


The Heap
Stack
array object
-1
2
index: 0 1
Frame for callByValue()
x: -1
arr
Frame for main()
x:19
arr

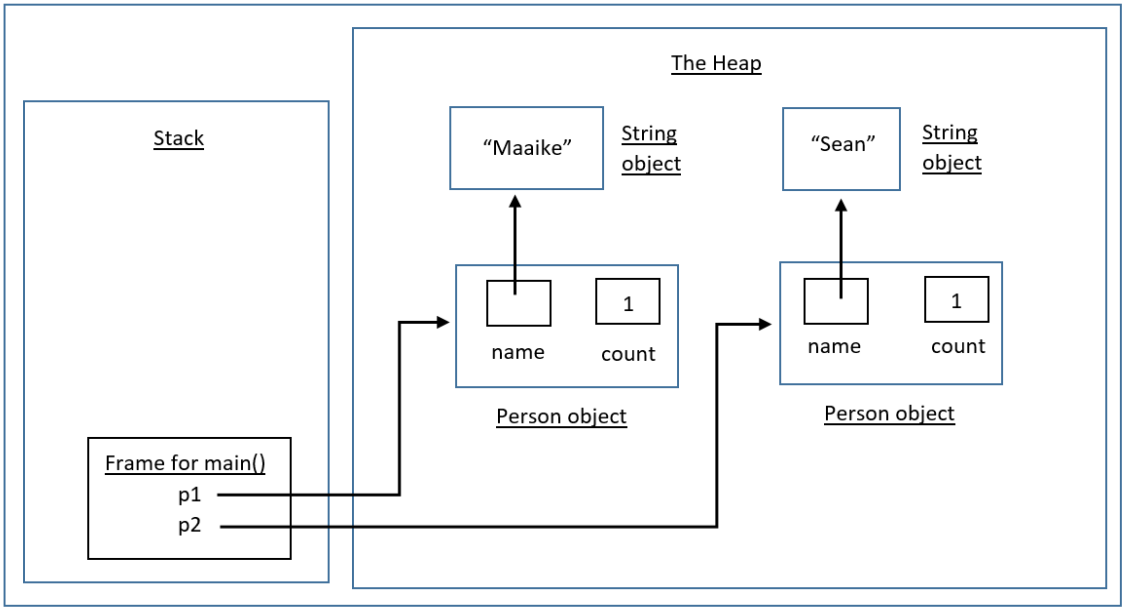
## Chapter 8: Classes, Objects, and Enums

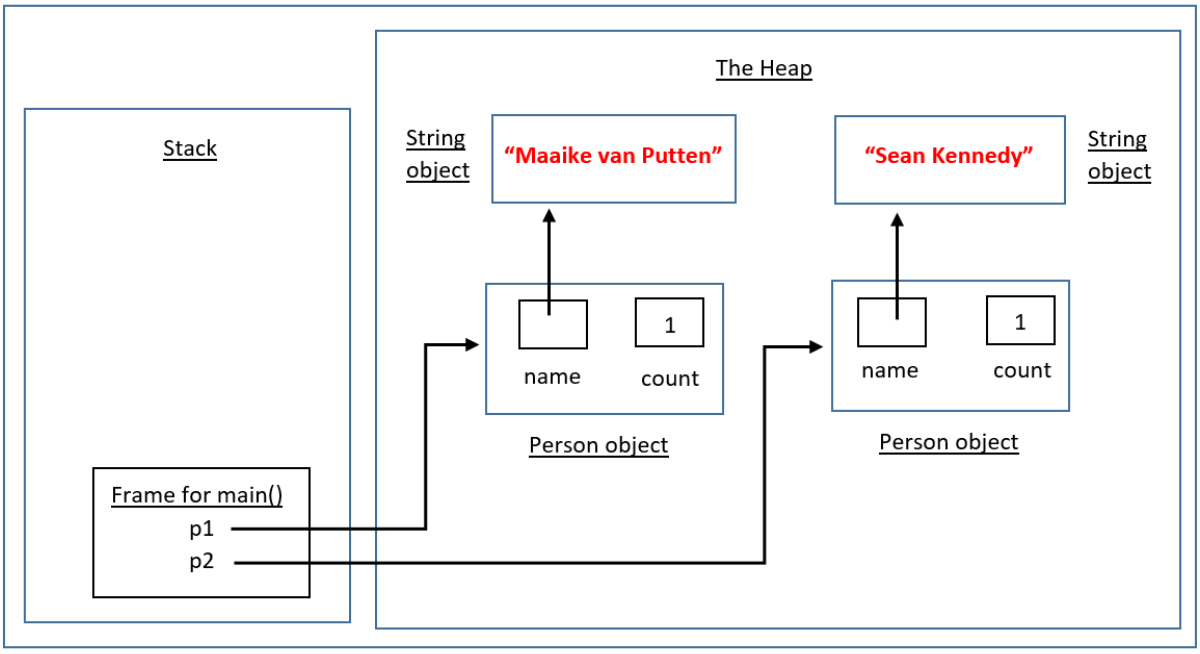
```
3   class Person{}
4   public class PersonExample {
5       public static void main(String[] args) {
6           Person p = new Person();
7       }
8   }
```

```
3   class Person{
4       Person(){
5           super();
6       }
7   }
8   public class PersonExample {
9       public PersonExample(){
10          super();
11      }
12      public static void main(String[] args) {
13          Person p = new Person();
14      }
15  }
```

```java
class Person{
    private String name;    // instance variable
    private int count;      // instance variable

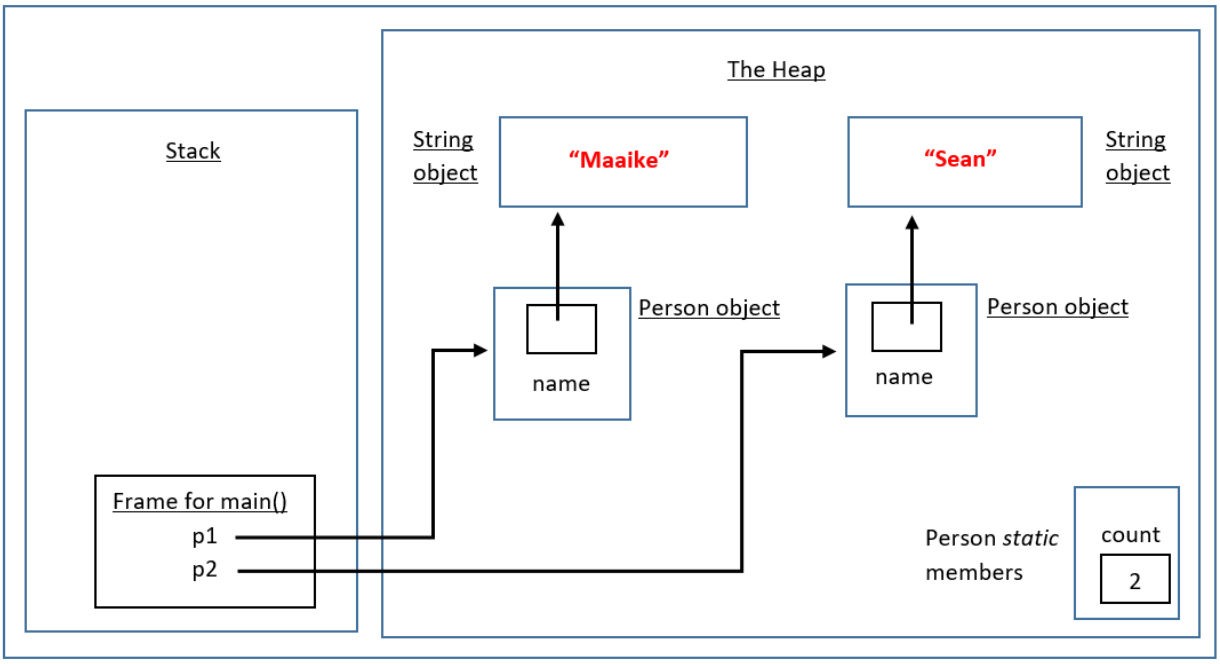
    Person(String aName) { // constructor
        name = aName;
        count++;
    }
    public String getName() { // instance method
        return name;
    }
    public void setName(String aName) { // instance method
        name = aName;
    }
    public int getCount() { // instance method
        return count;
    }
}
public class PersonExample {
    public static void main(String[] args) {
        Person p1 = new Person( aName: "Maaike");
        Person p2 = new Person( aName: "Sean");
        System.out.println(p1.getName()); // Maaike
        System.out.println(p2.getName()); // Sean
        p1.setName("Maaike van Putten");
        p2.setName("Sean Kennedy");
        System.out.println(p1.getName()); // Maaike van Putten
        System.out.println(p2.getName()); // Sean Kennedy
    }
}
```


The Heap
Stack
"Maaike"
String object
"Sean"
String object
1
name
count
1
name
count
Person object
Person object
Frame for main()
p1
p2


The Heap
Stack
String object
"Maaike van Putten"
"Sean Kennedy"
String object
1
name
count
1
name
count
Person object
Person object
Frame for main()
p1
p2

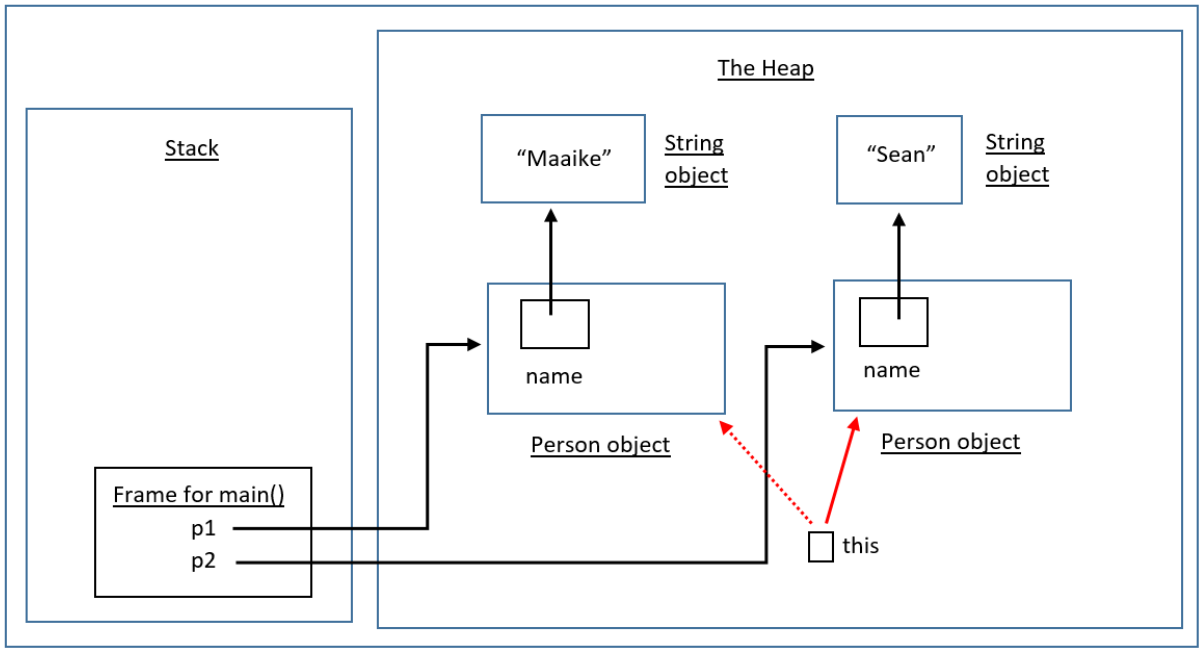
```java
class Person{
    private String name;        // instance variable
    private static int count;   // class variable

    Person(String aName) { // constructor
        name = aName;
        Person.count++;
    }
    public String getName() { // instance method
        return name;
    }
    public void setName(String aName) { // instance method
        name = aName;
    }
    public static int getCount() { // class method
        return Person.count;
    }
}
public class PersonExample {
    public static void main(String[] args) {
        Person p1 = new Person( aName: "Maaike");
        Person p2 = new Person( aName: "Sean");
        System.out.println(Person.getCount()); // 2
    }
}
```


The Heap
Stack
String object
"Maaike"
"Sean"
String object
Person object
name
Person object
name
Frame for main()
p1
p2
Person static members
count
2

```java
class Person{
    private String name;    // instance variable

    Person(String aName) { // constructor
//        name = aName;
        this.name = aName;
    }
    public String getName() { // instance method
//        return name;
        return this.name;
    }
    public void setName(String aName) { // instance method
//        name = aName;
        this.name = aName;
    }
}
public class PersonExample {
    public static void main(String[] args) {
        Person p1 = new Person( aName: "Maaike");
        Person p2 = new Person( aName: "Sean");
        System.out.println(p1.getName()); // Maaike
        System.out.println(p2.getName()); // Sean
        p1.setName("Maaike van Putten");
        p2.setName("Sean Kennedy");
        System.out.println(p1.getName()); // Maaike van Putten
        System.out.println(p2.getName()); // Sean Kennedy
    }
}
```




```java
class Person{
    private String name;    // instance variable

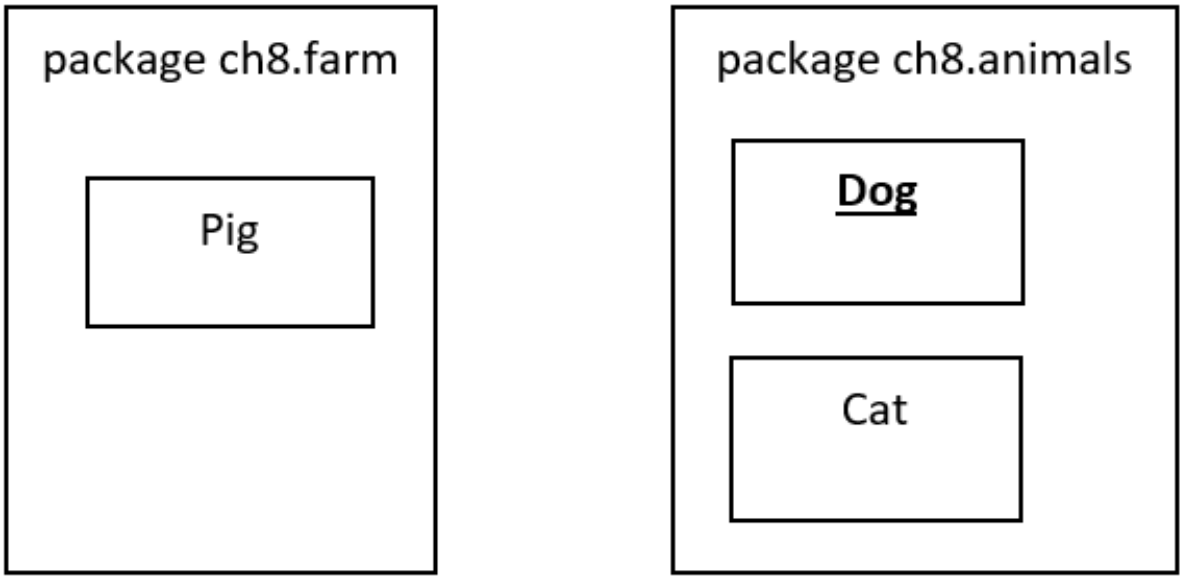
    Person(String name) { // constructor
        name = name; // shadowing/hiding the instance variable
    }
    public String getName() { // instance method
        return name;
    }
    public void setName(String aName) { // instance method
        name = aName;
    }
}
public class PersonExample {
    public static void main(String[] args) {
        Person p1 = new Person( name: "Maaike");
        System.out.println(p1.getName()); // null
    }
}
```

```java
class Person{
    private String name;    // instance variable

    Person(String name) { // constructor
        this.name = name;
    }
    public String getName() { // instance method
        return name;
    }
    public void setName(String aName) { // instance method
        name = aName;
    }
}
public class PersonExample {
    public static void main(String[] args) {
        Person p1 = new Person( name: "Maaike");
        System.out.println(p1.getName()); // Maaike
    }
}
```

```java
public class PersonExample {
    int x;              // instance variable
    public void m(){}   // instance method
    public static void main(String[] args) {
        // Non-static field 'x' cannot be referenced
        // from a static context
        x = 9;  // same as 'this.x = 9;'
        this.x = 99;
        m();    // same as 'this.m();'
        this.m();

        // this works
        PersonExample pe = new PersonExample();
        pe.x=999;   // ok
        pe.m();     // ok
        System.out.println(pe.x);   // 999
    }
}
```



```java
package ch8.animals;

public class Dog {
    private String dogName;
    protected int age;
    public Dog(String dogName) { this.dogName = dogName; }
    public String getDogName() { return dogName; }
    void pkgPrivate(){}
}
class Cat{  // package private
    Cat(){}
    public void testDogAccess(){
        Dog d = new Dog( dogName: "Rex");
//        d.dogName = "Abc"; // dogName is private to Dog
        d.age = 2;
        d.pkgPrivate(); //ok
    }
}
```

```java
package ch8.farm;

import ch8.animals.Dog; // class is public, ok
//import ch8.animals.Cat; // class is pkg-private, error

public class Pig{
    void testDog(){
        Dog d = new Dog( dogName: "Shep"); // constructor is public
//        d.pkgPrivate(); // package-private method, error
    }
}
```

```java
package ch8;

class Adult{
    private String name;
    private int age;

    Adult(String name, int age) {
        this.age = age;
        this.name = name;
    }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}
public class BasicEncapsulation {
    public static void main(String[] args) {
        Adult john = new Adult( name: "John",  age: 20);
        System.out.println(john.getName() + " "
                + john.getAge());   // John 20
        //john.age = -99; // 'age' is private
        john.setAge(-99); // uh-oh!
        System.out.println(john.getName() + " "
                + john.getAge());   // John -99
    }
}
```

```java
package ch8;

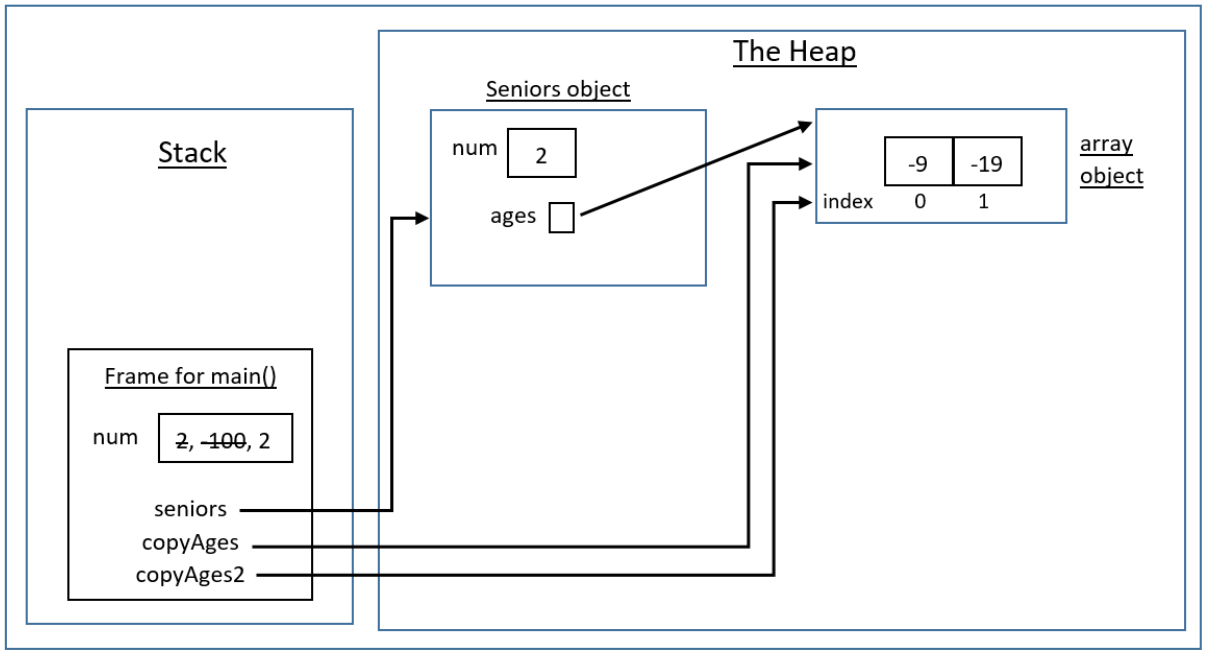
class Adult{
    private String name;
    private int age;

    Adult(String name, int age) {
        setAge(age);
        this.name = name;
    }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getAge() { return age; }
    public void setAge(int age) {
        if(isAgeOk(age)){
            this.age = age;
        }else{
            this.age = -1; // error state
        }
    }
    private boolean isAgeOk(int age){ // private
        return age >= 18 ? true :false ; // ternary operator
    }
}
```

```java
class Seniors {
    private int[] ages = new int[2];
    private int num;

    Seniors(){
        num = 2;
        ages[0] = 30;
        ages[1] = 40;
    }
    public int getNum() { return num;}
    public int[] getAges() { // breaks encapsulation
        return ages;
    }
}
public class AdvancedEncapsulation {
    public static void main(String[] args) {
        Seniors seniors = new Seniors();
        // 1. Returning primitives is okay.
        int num = seniors.getNum();
        System.out.println(num); // 2
        num = -100;
        num = seniors.getNum();
        System.out.println(num); // 2, ok, primitives are encapsulated once 'private'

        // 2. Returning references requires care.
        int[] copyAges = seniors.getAges(); // 'copyAges' and 'ages' refer to the same array object!
        System.out.println(copyAges[0] + ", " + copyAges[1]);// 30, 40
        // As we have a copy of the internal array reference, we can, from HERE
        // change the "private" internal Seniors array! This breaks encapsulation.
        copyAges[0] = -9;
        copyAges[1] = -19;
        int[] copyAges2 = seniors.getAges();
        System.out.println(copyAges2[0] + ", " + copyAges2[1]);// -9, -19
    }
}
```

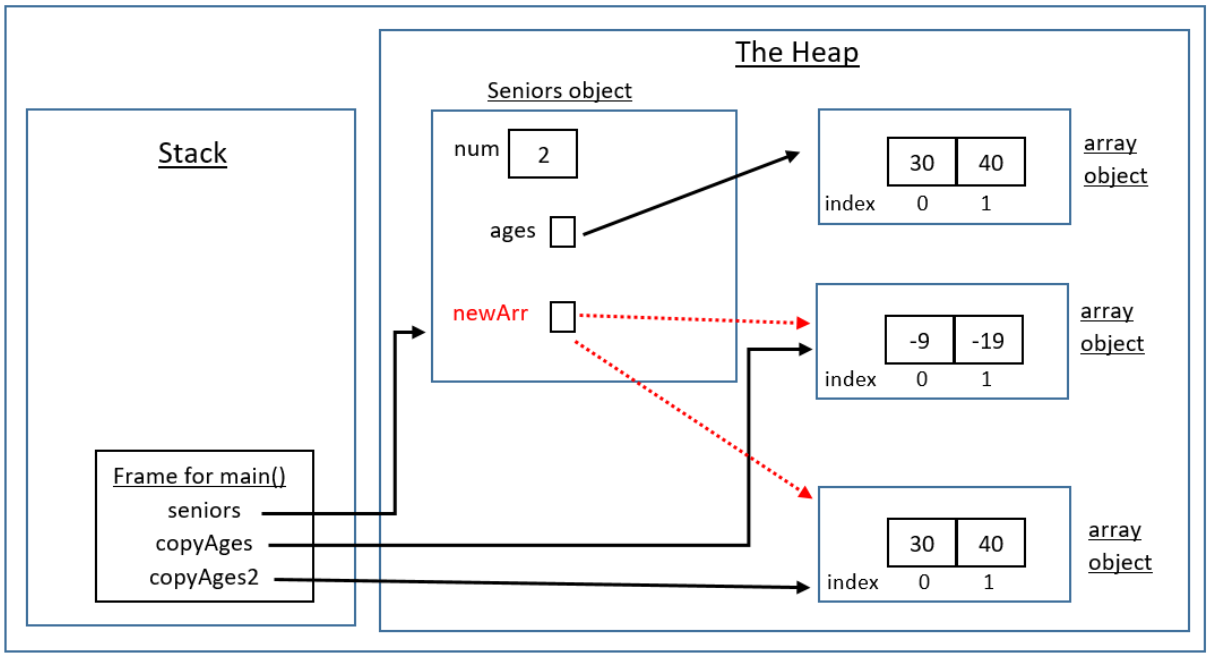

The Heap
Seniors object
Stack
num
2
ages
-9
-19
index
0
1
array
object
Frame for main()
num
2, -100, 2
seniors
copyAges
copyAges2

```java
import java.util.Arrays;

class Seniors {
    private int[] ages = new int[2];
    private int num;

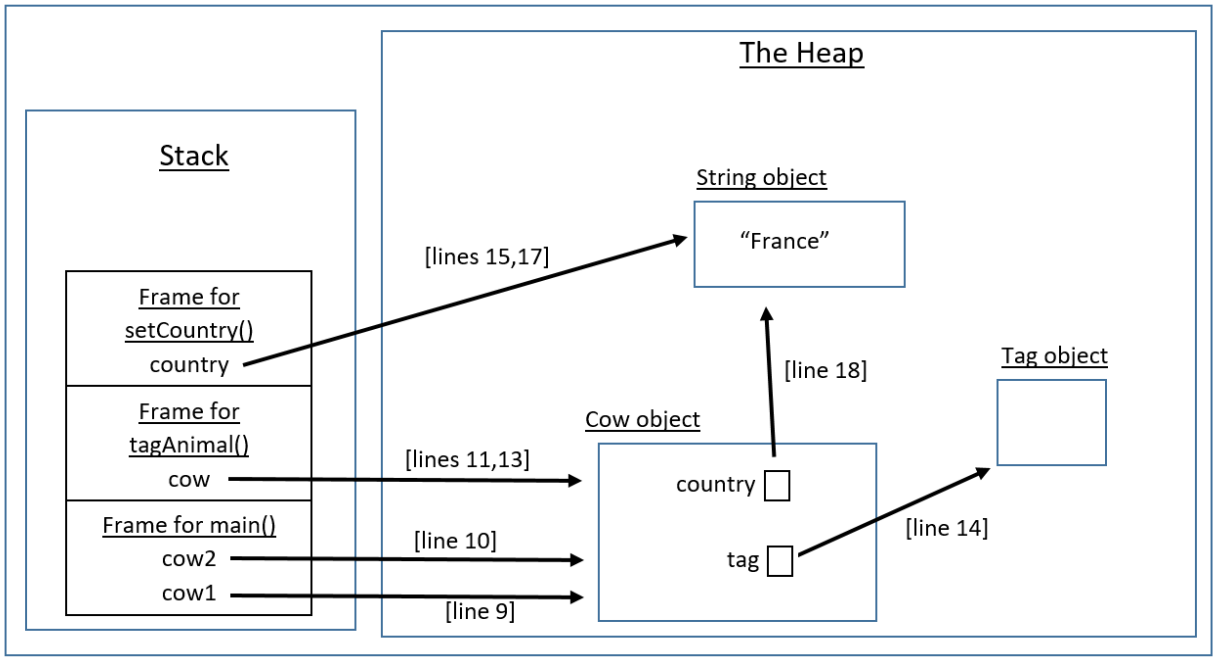
    Seniors(){
        num = 2;
        ages[0] = 30;
        ages[1] = 40;
    }
    public int getNum() { return num;}
    public int[] getAges() { // properly encapsulated
        int newArr[] =  Arrays.copyOf(ages, newLength: 2);
        return newArr;
    }
}
public class AdvancedEncapsulation {
    public static void main(String[] args) {
        Seniors seniors = new Seniors();

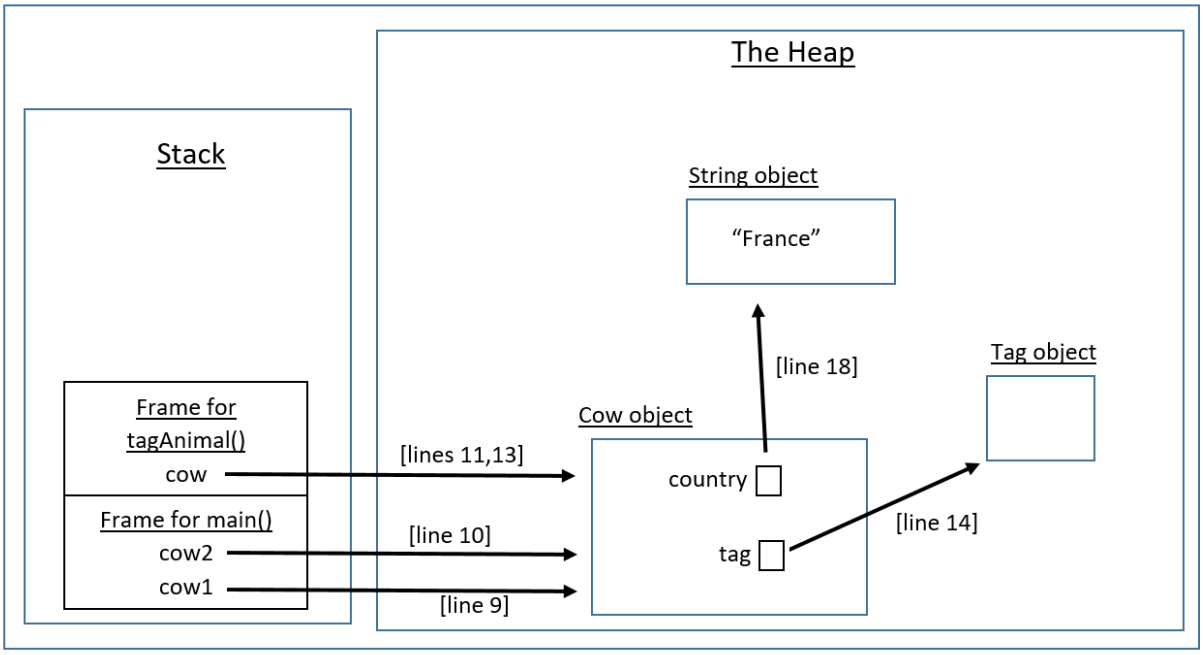
        int[] copyAges = seniors.getAges(); // 'copyAges' and 'ages' refer to 2 different arrays
        System.out.println(copyAges[0] + ", " + copyAges[1]);// 30, 40
        copyAges[0] = -9;
        copyAges[1] = -19;
        int[] copyAges2 = seniors.getAges();
        System.out.println(copyAges2[0] + ", " + copyAges2[1]);// 30, 40
    }
}
```

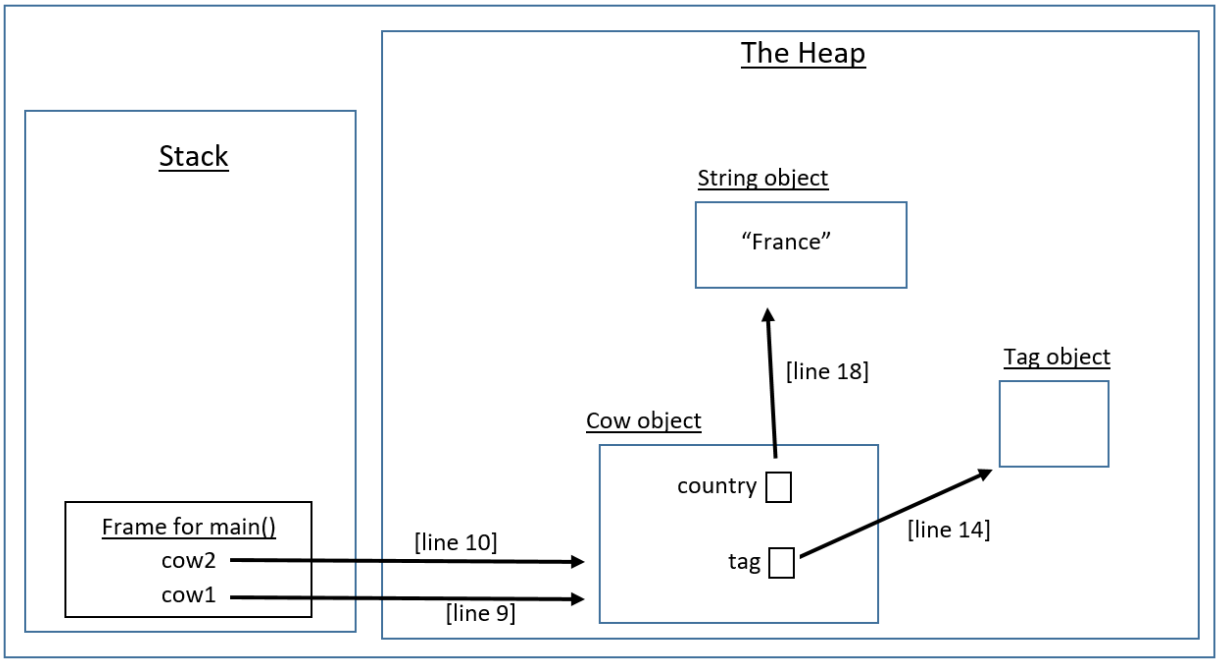



```java
class Tag{}
public class Cow {
    Tag tag;
    String country;

    public static void main(String[] args) {
        Cow cow1 = new Cow();
        Cow cow2 = cow1;// reassignment
        cow2.tagAnimal(cow1);
    }
    void tagAnimal(Cow cow){
        tag = new Tag();
        cow.setCountry("France");
    }
    void setCountry(String country){
        this.country = country;
    }
}
```


The Heap
Stack
String object
"France"
[lines 15,17]
Frame for setCountry()
country
[line 18]
Tag object
Frame for tagAnimal()
cow
Cow object
[lines 11,13]
country
Frame for main()
cow2
[line 10]
tag
[line 14]
cow1
[line 9]


The Heap
Stack
String object
"France"
[line 18]
Tag object
Frame for tagAnimal()
cow
Cow object
[lines 11,13]
country
Frame for main()
cow2
[line 10]
tag
[line 14]
cow1
[line 9]




```
3   class Dog{}
4   class Cat{}
5   public class InstanceOfExample {
6       public static void main(String[] args) {
7           Dog dog = new Dog();
8           Cat cat = new Cat();
9           if(dog instanceof Dog){ // true
10              System.out.println("dog referring to a Dog object");
11          }
12          if(cat instanceof Cat){ // true
13              System.out.println("cat referring to a Cat object");
14          }
15  //          if(cat instanceof Dog){ // Cat and Dog are completely separate classes
16  //              System.out.println("cat referring to a Dog object");
17  //          }
18      }
19  }
```

```java
enum Water {
    STILL, SPARKLING;
}
public class SimpleEnums {
    public static void main(String[] args) {
//        Water stillWater = new Water();// compiler error
//        Water stillWater = Water.EXTRA_SPARKLING; // type safety

        Water stillWater = Water.STILL;
        System.out.println(stillWater == Water.STILL);        // true
        System.out.println(stillWater.equals(Water.STILL));   // true
        switch(stillWater){
            case STILL:
                System.out.println("Still water");
                break;
//            case Water.STILL: // unqualified enum value required
//            case 0: // cannot use an int
        }
//        if(Water.STILL == 0){}// Water == int
        Water sparklingWater = Water.valueOf( name: "SPARKLING");
        System.out.println(sparklingWater);// SPARKLING

        for(Water water: Water.values()){
            // Ordinal value of: 0 is STILL
            // Ordinal value of: 1 is SPARKLING
            System.out.println("Ordinal value of: "+ water.ordinal() + " is "+ water.name());
        }

    }
}
```

```java
enum WorkDay {
    // values must be first
    MONDAY( hoursOfWork: "9-5"),// constructor calls
    TUESDAY( hoursOfWork: "9-5"),
    WEDNESDAY( hoursOfWork: "9-5"),
    THURSDAY( hoursOfWork: "9-5"),
    FRIDAY( hoursOfWork: "9-5"),
    SATURDAY( hoursOfWork: "10-1"){
        // constant specific class body
        public String getWorkLocation(){ return "Home";}
    };// ; required at end

    private String hoursOfWork;
    WorkDay(String hoursOfWork) {// constructor is 'private'
        this.hoursOfWork = hoursOfWork;
    }
    public String getHoursOfWork() {
        return hoursOfWork;
    }
    public String getWorkLocation() {
        return "Office";
    }
}
public class ComplexEnums {
    public static void main(String[] args) {
         WorkDay monday = WorkDay.MONDAY;
         System.out.println(monday.getHoursOfWork()+", "   // 9-5,
                 +monday.getWorkLocation());               // Office
         System.out.println(WorkDay.SATURDAY.getHoursOfWork() + ", "  // 10-1
                 +WorkDay.SATURDAY.getWorkLocation());                // Home
    }
}
```

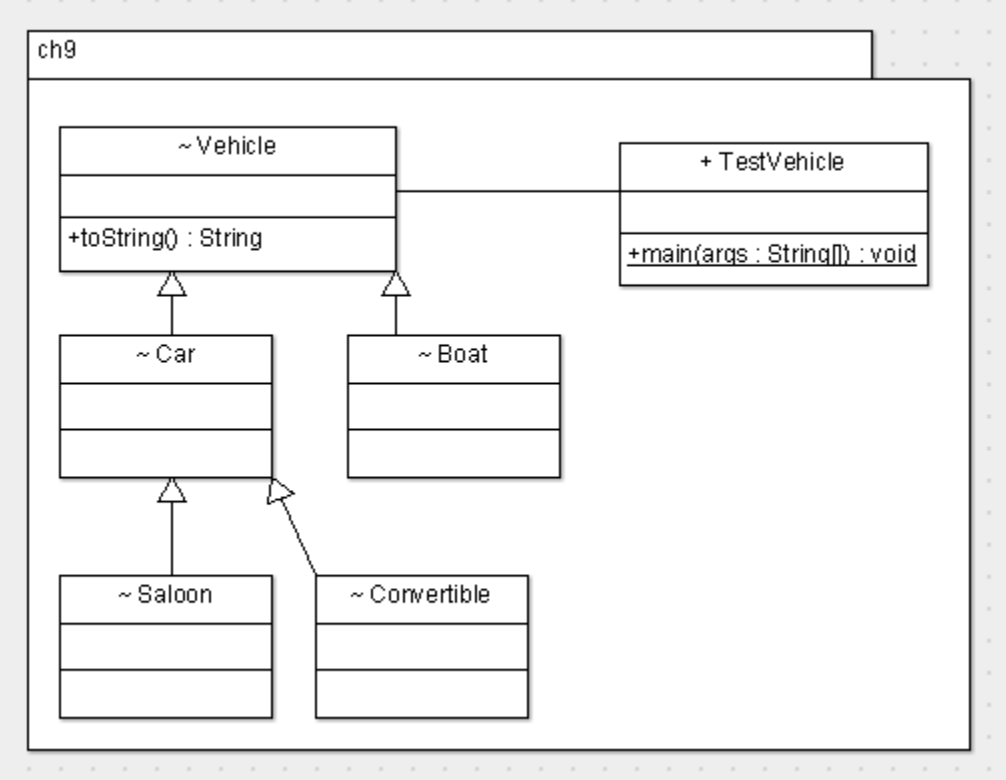
```java
public final class Person {
    private final String name;
    private final Integer age;

    public Person(String name, Integer age) {
        this.name = name;
        this.age = age;
    }
    public String name() {
        return name;
    }
    public Integer age() {
        return age;
    }
    @Override
    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (obj == null || obj.getClass() != this.getClass()) return false;
        var that = (Person) obj;
        return Objects.equals(this.name, that.name) &&
                Objects.equals(this.age, that.age);
    }
    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
    @Override
    public String toString() {
        return "Person[" +
                "name=" + name + ", " +
                "age=" + age + ']';
    }
}
```

```java
public record Person(String name, Integer age) { }
```

```java
public record Person(String name, Integer age) {
    // compact canonical constructor
//    public Person(String name, Integer age)  {
//        if(age < 18){
//            this.name = "Error"; this.age = -1;
//        }
//        this.name = name;
//        this.age  = age;
//    }
        // compact constructor
        public Person  {
            if(age < 18){
                name = "Error"; age = -1;
            }
        }
}
class PersonTest{
    public static void main(String[] args) {
        Person p1 = new Person( name: "Joe Bloggs",  age: 20);
        System.out.println(p1);         // Person[name=Joe Bloggs, age=20]
        System.out.println(p1.name());  // Joe Bloggs
        System.out.println(p1.age());   // 20
    }
}
```

# Chapter 9: Inheritance and Polymorphism

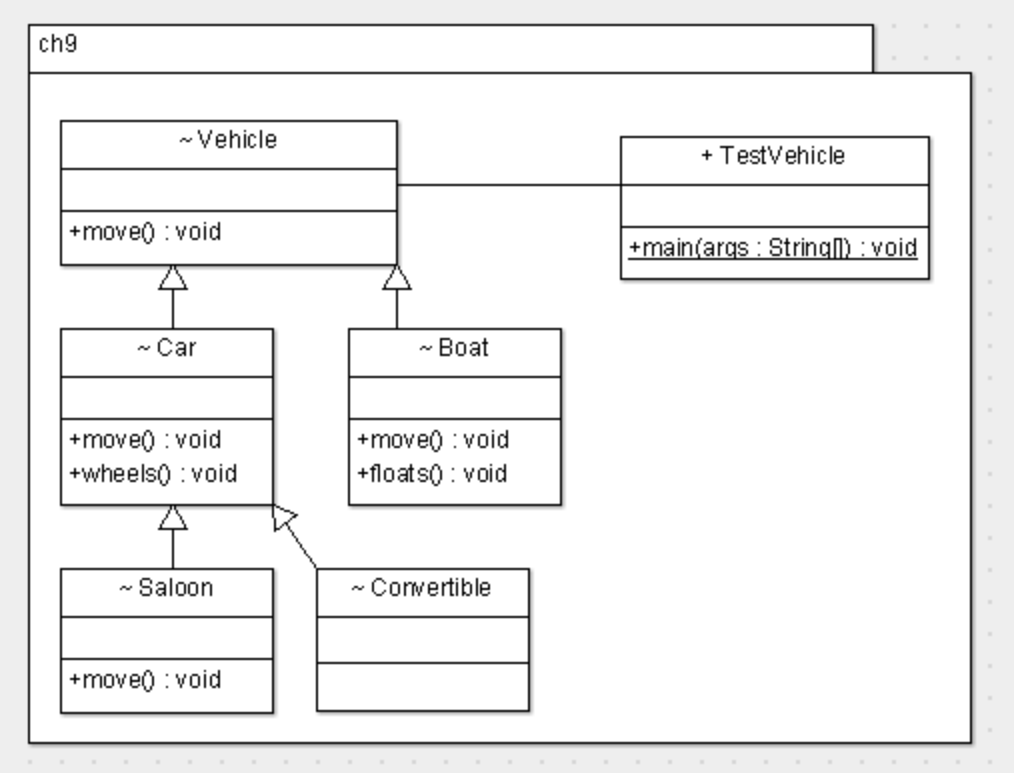


```java
package ch9;

// class Vehicle extends Object
class Vehicle{
    public String toString(){
        return "Vehicle::toString()";
    }
}
class Car extends Vehicle{}
class Boat extends Vehicle{}
class Saloon extends Car {}
class Convertible extends Car {}

public class TestVehicle {
    public static void main(String[] args) {
        Vehicle vehicle = new Vehicle();
        System.out.println(vehicle.toString()); // Vehicle::toString()
        Car     car     = new Car();
        // next line invokes car.toString()
        System.out.println(car);                // Vehicle::toString()
        Saloon  saloon  = new Saloon();
        System.out.println(saloon);             // Vehicle::toString()

        System.out.println(new TestVehicle().toString());// ch9.TestVehicle@378bf509
    }
}
```


ch9
~ Vehicle
+move() : void
+ TestVehicle
+main(args : String[]) : void
~ Car
+move() : void
+wheels() : void
~ Boat
+move() : void
+floats() : void
~ Saloon
+move() : void
~ Convertible

```java
class Vehicle{
    public void move(){ System.out.println("Vehicle::move"); }
}
class Car extends Vehicle{
    @Override public void move(){ System.out.println("Car::move()"); }
    public void wheels(){ System.out.println("Car::wheels()"); }
}
class Boat extends Vehicle{
    @Override public void move(){ System.out.println("Boat::move()"); }
    public void floats(){ System.out.println("Boat::floats()"); }
}
class Saloon extends Car {
    @Override public void move(){ System.out.println("Saloon::move()"); }
}
class Convertible extends Car {}

public class TestVehicle {
    public static void main(String[] args) {
        Vehicle v = new Car();
        v.move();   // Car::move()
        v = new Boat();
        v.move();   // Boat::move()
//        v.floats(); // compiler error
        v = new Saloon();
        v.move();   // Saloon::move()
        v = new Convertible();
        v.move();   // Car::move()

        Saloon s = (Saloon)new Vehicle(); // ClassCastException
    }
}
```

```java
class Vehicle{
    public void move(){ System.out.println("Vehicle::move"); }
}
class Car extends Vehicle{
    @Override public void move(){ System.out.println("Car::move()"); }
    public void wheels(){ System.out.println("Car::wheels()"); }
}
class Boat extends Vehicle{
    @Override public void move(){ System.out.println("Boat::move()"); }
    public void floats(){ System.out.println("Boat::floats()"); }
}
class Saloon extends Car {
    @Override public void move(){ System.out.println("Saloon::move()"); }
}
class Convertible extends Car {}

public class TestVehicle {
    public static void doAction(Vehicle v){
        v.move();
    }
    public static void main(String[] args) {
        Vehicle v = new Car();
        doAction(v);                // Car::move()
        doAction(new Boat());       // Boat::move()
        doAction(new Saloon());     // Saloon::move()
        doAction(new Convertible());// Car::move()
    }
}
```

```java
class Vehicle{
    double cost = 100.0;        // instance data
    static int age = 1;         // class data
    public void move(){         // instance method
        System.out.println("Vehicle::move()");
    }
    public static void sm() {   // class method
        System.out.println("Vehicle::sm()");
    }
}
class Car extends Vehicle{
    double cost = 20_000.0; // hiding
    static int age = 2;     // hiding

    @Override public void move(){   // overriding
        System.out.println("Car::move()");
    }
    public static void sm() {       // hiding
        System.out.println("Car::sm()");
    }
}
public class TestVehicle {
    public static void main(String[] args) {
        Vehicle v = new Car();
        System.out.println(v.cost); // 100.0
        System.out.println(v.age);  // 1
        v.sm();                     // Vehicle::sm()
        v.move();                   // Car::move()
    }
}
```

```java
class Animal{
    public void eat(){}
}
class Cow extends Animal{
    public void eat(){}         // overriding, same signature
    public void eat(String s){} // overloaded, different signature!
}
public class OverloadTest {
    public void calc(int x, double y){} //  calc(int, double)
    public void calc(){}                //  calc()
    public void calc(int x){}           //  calc(int)
    public void calc(double y){}        //  calc(double)
    public void calc(double y, int x){} //  calc(double, int)

//    public void calc(int a, double b){}           // calc(int, double)
//    public int calc(int a, double b){ return 1; } // calc(int, double)

    public static void main(String[] args) {
        Animal aa = new Animal();
        aa.eat();
//        aa.eat("Grass"); // compiler error

        Animal ac = new Cow();
        ac.eat();
//        ac.eat("Grass"); // compiler error

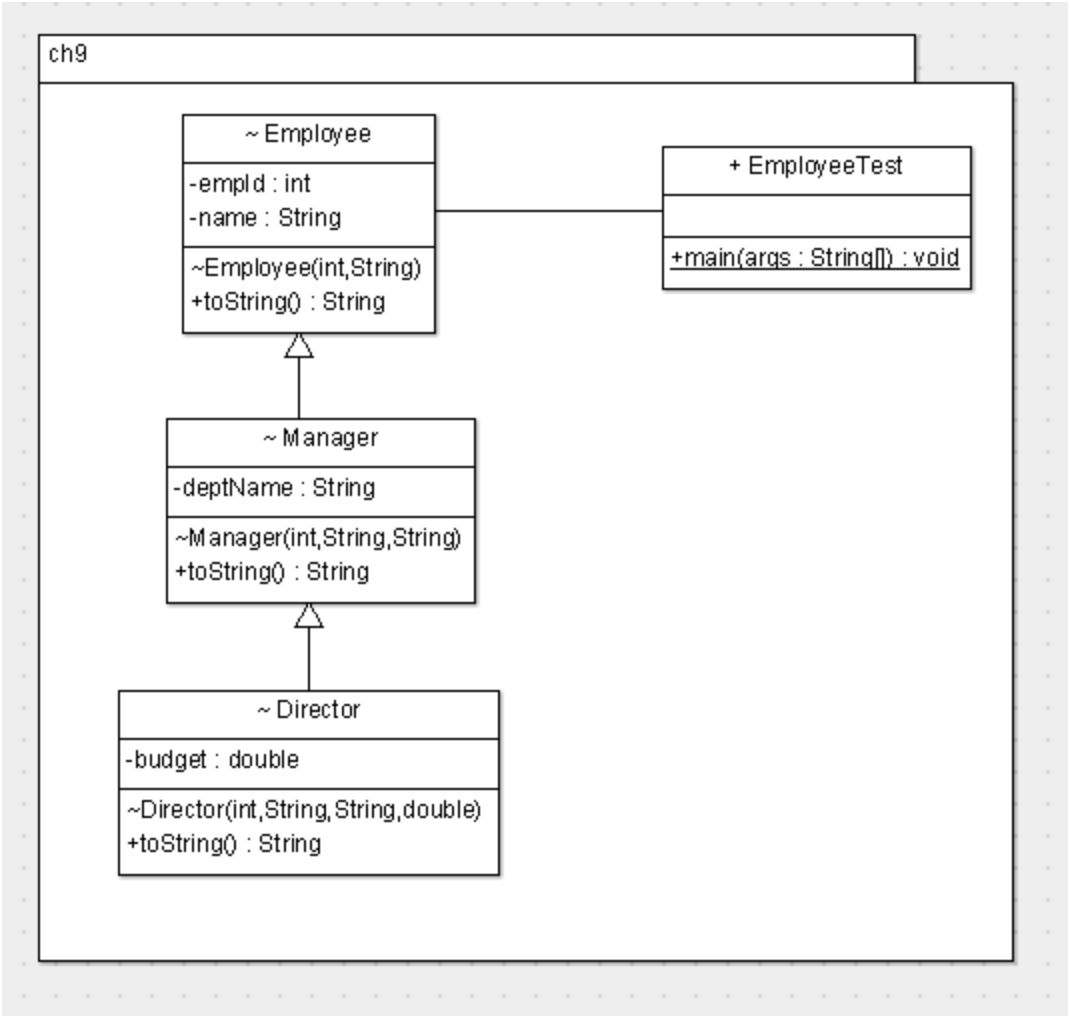
        Cow cc = new Cow();
        cc.eat();             // inherited
        cc.eat( s: "Grass");
    }
```

```java
import java.io.IOException;

class Dog{
    public void walk(){System.out.println("Dog::walk()");};
    public Dog run() { return new Dog(); }
}
class Terrier extends Dog{
//    public String walk(){ return "Walk the Dog";} // return type should be void
//    private void walk(); // access rights cannot be weaker
//    public void walk() throws IOException {} // cannot throw new checked exceptions
    public void walk(int metres){} // an overload, not an override
    @Override public void walk(){System.out.println("Terrier::walk()");};

//    @Override public Dog run() {return new Dog();}            // ok
//    @Override public Terrier run() {return new Terrier();}    // ok
    @Override public Dog run() {return new Terrier();}          // ok
}

public class OverridingTest {
    public static void main(String[] args) {
        Dog dt = new Terrier();
        dt.walk();  // Terrier::walk()
        Dog d = dt.run();
        if(d instanceof Terrier){
            System.out.println("Terrier object!");  // Terrier object!
        }
    }
}
```

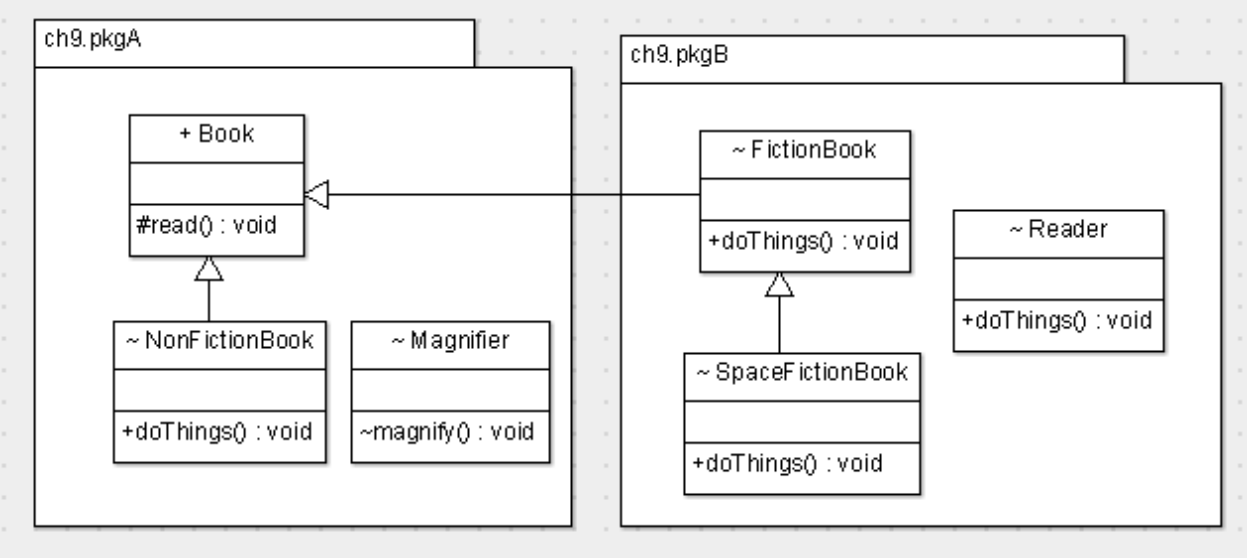

ch9
~ Employee
-empId : int
-name : String
~Employee(int,String)
+toString() : String
+ EmployeeTest
+main(args : String[]) : void
~ Manager
-deptName : String
~Manager(int,String,String)
+toString() : String
~ Director
-budget : double
~Director(int,String,String,double)
+toString() : String

```java
class Employee {
    private int empId;
    private String name;

    Employee(int empId, String name) {
        this.empId = empId;
        this.name = name;
    }
    @Override public String toString() { return "ID: " + empId + ", " +  "Name: " + name + ","; }
}
class Manager extends Employee {// a Manager "IS-A" Employee
    private String deptName;    // a Manager "HAS-A" department

    Manager(int empId, String name, String deptName) {
        super(empId, name); // call parent constructor
        this.deptName = deptName;
    }
    @Override
    public String toString() {
        // call the parent toString()
        return super.toString() + " Department: " + deptName + ",";
    }
}
class Director extends Manager {
    private double budget;

    Director(int empId, String name, String department, double budget) {
        super(empId, name, department);
        this.budget = budget;
    }
    @Override public String toString() { return super.toString() + " Budget: " + budget; }
}
public class EmployeeTest {
    public static void main(String[] args) {
        Employee emplDir = new Director( empId: 754,  name: "Joe Bloggs",  department: "Marketing",  budget: 10_000.00);
        System.out.println(emplDir); // ID: 754, Name: Joe Bloggs, Department: Marketing, Budget: 10000.0
    }
}
```




```java
package ch9.pkgA;

public class Book {
    protected void read(){}
}
class NonFictionBook extends Book{
    public void doThings(){
        read(); // same package; no problem
    }
}
class Magnifier{
    void magnify(){
        Book b = new Book();
        b.read(); // same package; no problem
    }
}
```

```java
package ch9.pkgB;

import ch9.pkgA.Book;

class FictionBook extends Book{
    public void doThings(){
        read();     // different package, via inheritance; no problem
        this.read();// different package, via inheritance; no problem
        FictionBook fb = new FictionBook(); // default ctor created for us
        fb.read();  // no problem

        // Here, I create an instance of the superclass that has the protected
        // member. Note that even though FictionBook has access via inheritance
        // to read(), FictionBook must access it properly.
        Book b = new Book();
        b.read();   // not public!
    }
}
class SpaceFictionBook extends FictionBook{
    public void doThings(){
        read();     // different package, via inheritance; no problem
        new Book().read();
        new FictionBook().read();
        new SpaceFictionBook().read();// ok
    }
}
class Reader{
    public void doThings(){
        Book b = new Book();
        b.read();

        // can I access the protected member via the subclass that inherits it?
        FictionBook fb = new FictionBook();
        fb.read();
    }
}
```

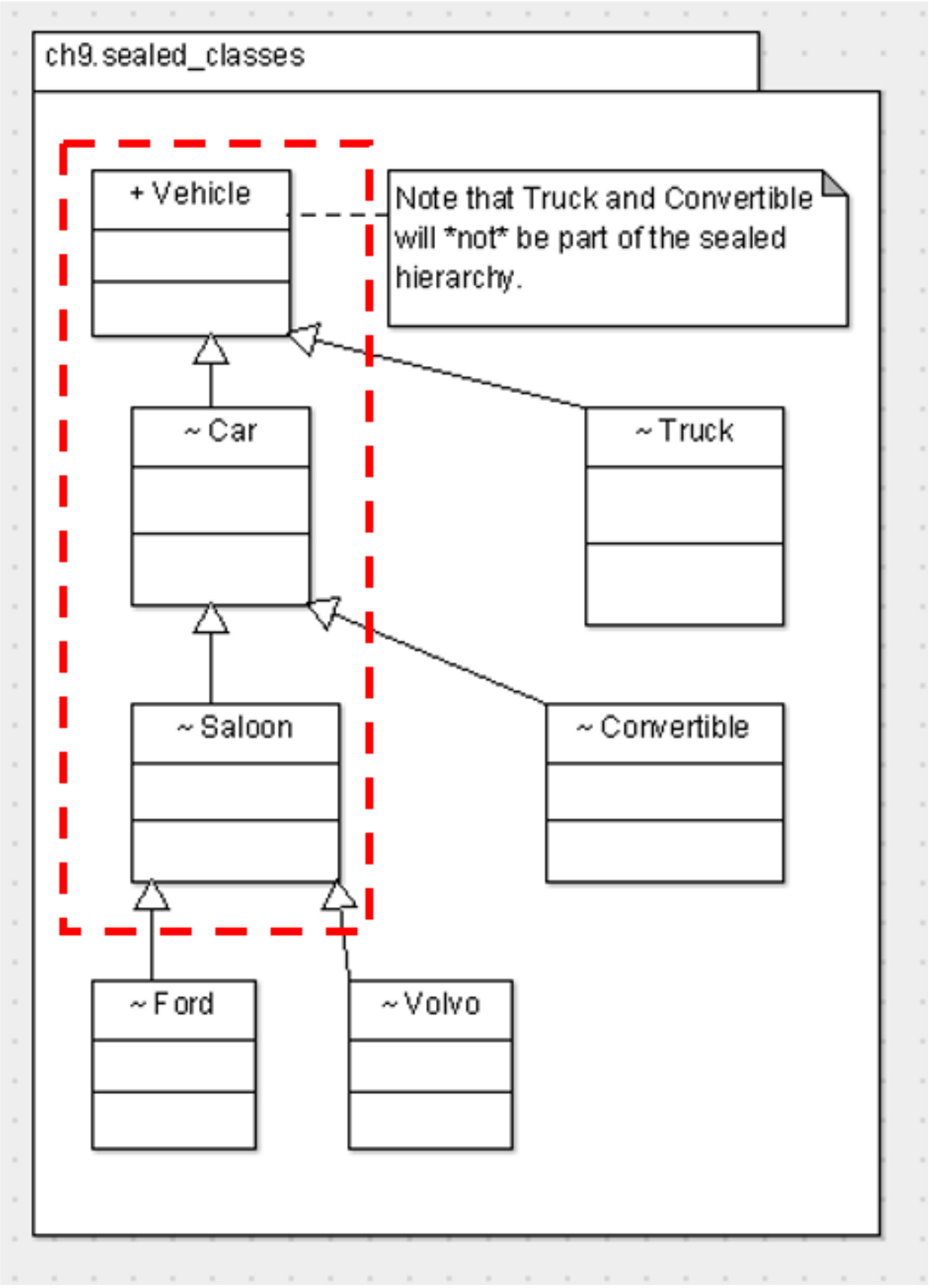
```java
abstract class Pencil{
    abstract void write(); // no {}
}
class CharcoalPencil extends Pencil{}
abstract class WaterColorPencil extends Pencil{}
class GraphitePencil extends Pencil{
    @Override
    void write(){
        System.out.println("GraphitePencil::write()");
    }
}

public class PencilsExample{
    public static void main(String[] args) {
        Pencil pp    = new Pencil(); // cannot "new" a Pencil (abstract)
        Pencil pdp   = new GraphitePencil();
        pdp.write();    // GraphitePencil::write()
    }
}
```

```java
final class Earth{}
// cannot extend a 'final' class
class SubEarth extends Earth{}

class Pen{
    final void write(){}
    // 'final' and 'abstract' not allowed together
    // as they have opposite meanings
//    final abstract scribble();
}
class FountainPen extends Pen{
    // cannot override a 'final' method
    @Override void write(){}
}
public class DemoOfFinal {
    final int ONE_YEAR = 1;
    void print(final String name, final int age){
        // primitives
        age = age + ONE_YEAR;
        // references - ok to access the object
        System.out.println(name.toUpperCase());
        // references - cannot modify the reference
        name = "Alexander";

        ONE_YEAR = 2; // cannot change a constant
    }
}
```


ch9.sealed_classes
+ Vehicle
Note that Truck and Convertible will *not* be part of the sealed hierarchy.
~ Car
~ Truck
~ Saloon
~ Convertible
~ Ford
~ Volvo

```java
public sealed class Vehicle permits Car{ }  // scoping hierarchy
sealed class Car extends Vehicle permits Saloon {}
//sealed class Truck extends Vehicle {}     // compiler error
non-sealed class Saloon extends Car{}       // opening up hierarchy again
class Volvo extends Saloon{}
class Ford extends Saloon{}
//class Convertible extends Car{}           // compiler error
```

```java
class Parent{
    // instance initialization block
    { System.out.println("6. Parent instance init block 1"); }
    // static initialization block
    static {System.out.println("2. Parent static init block 1");}
    Parent() { System.out.println("8. Parent constructor"); }
    { System.out.println("7. Parent instance init block 2"); }
    static {System.out.println("3. Parent static init block 2");}
}
class Child extends Parent{
    { System.out.println("9. Child instance init block 1"); }
    static {System.out.println("4. Child static init block 1");}
    Child() { System.out.println("11. Child constructor"); }
    { System.out.println("10. Child instance init block 2"); }
    static {System.out.println("5. Child static init block 2\n");}
}
public class InitializationBlocks {
    static {System.out.println("1. InitializationBlocks static init block");}
    { System.out.println("InitializationBlocks instance init block"); }
    public static void main(String[] args) {
        System.out.println("---> Creating first Child object...");
        new Child();
        System.out.println("\n--->Creating second Child object...");
        new Child();
    }
}
```

```
1. InitializationBlocks static init block

--->Creating first Child object...

2. Parent static init block 1
3. Parent static init block 2
4. Child static init block 1
5. Child static init block 2

6. Parent instance init block 1
7. Parent instance init block 2
8. Parent constructor
9. Child instance init block 1
10.Child instance init block 2
11.Child constructor

--->Creating second Child object...

6. Parent instance init block 1
7. Parent instance init block 2
8. Parent constructor
9. Child instance init block 1
10.Child instance init block 2
11.Child constructor
```

```java
class Machine {
    void on(){ System.out.println("Machine::on()"); }
}
class Tractor extends Machine {
    @Override void on(){ System.out.println("Tractor::on()"); }
    void drive() { System.out.println("Tractor::drive()"); }
}
public class UpcastingAndDowncasting {
    public static void doAction(Machine machine){
        machine.on();
    }
    public static void main(String[] args) {
        Machine mt = new Tractor(); // upcasting
        doAction(mt);               // polymorphism, Tractor::on()
        doAction(new Tractor());    // polymorphism, Tractor::on()
    }
}
```

```java
class Machine {
    void on(){ System.out.println("Machine::on()"); }
}
class Tractor extends Machine {
    @Override void on(){ System.out.println("Tractor::on()"); }
    void drive() { System.out.println("Tractor::drive()"); }
}
public class UpcastingAndDowncasting {
    public static void doAction(Machine machine){
//        machine.on();

        // Let us try and call the Tractor-specific method 'drive()'
//        machine.drive(); // compiler error
//        ((Tractor)machine).drive(); // possible ClassCastException
        if(machine instanceof Tractor t){
            t.drive(); // safe
        }
    }
    public static void main(String[] args) {
        doAction(new Machine());    // outputs nothing
        doAction(new Tractor());    // Tractor::drive()
    }
}
```

```java
class Machine {
    void on(){ System.out.println("Machine::on()"); }
}
class Tractor extends Machine {
    @Override void on(){ System.out.println("Tractor::on()"); }
    void drive() { System.out.println("Tractor::drive()"); }
}
public class UpcastingAndDowncasting {
    public static void doAction(Machine machine){
//        machine.on();

        // Let us try and call the Tractor-specific method 'drive()'
//        machine.drive(); // compiler error
//        ((Tractor)machine).drive(); // possible ClassCastException
        if(machine instanceof Tractor t){
            t.drive(); // safe
        }
    }
    public static void main(String[] args) {
        doAction(new Machine());    // outputs nothing
        doAction(new Tractor());    // Tractor::drive()
    }
}
```

# Chapter 10: Interfaces and Abstract Classes

```java
interface I1{
    public abstract void m1();
    void m2();   // public abstract by default
    private void m3(){};
//    protected void m4(); // compiler error
}
```

```java
interface I2{
    public static final int VALUE1=1;
    int VALUE2=2; // public static final as well
}
```

```java
interface Moveable{
    String HOW="walk";// constant - public static final
    void move();      // public abstract by default
}
public class Dog implements Moveable{
    // MUST be public - cannot assign weaker privileges
//    void move(){}
    @Override
    public void move(){// MUST be public
        System.out.println("Dog::move()");
    }
    public static void main(String[] args) {
//        HOW = "walk"; // cannot change a final variable
        System.out.println(Moveable.HOW);// walk
        System.out.println(HOW);         // walk
       // cannot refer to an instance member from a static context
//        move();
       new Dog().move();                // Dog::move()
    }
}
```

```java
interface MoveableObject{} // tagging interface
interface Spherical{
    void doSphericalThings();
}
interface Bounceable extends MoveableObject, Spherical{
    void bounce();
}
// Concrete class Volleyball must implement all abstract
// methods in Bounceable
class Volleyball implements Bounceable{
    @Override public void doSphericalThings(){}
    @Override public void bounce(){}
}
// Abstract class Beachball is ok - can implement
// some, all or none of the abstract methods in Bounceable
abstract class Beachball implements Bounceable{}

public class InterfaceInheritance {
}
```

```java
package ch10.defaultMethods;

interface Moveable{
//    default void m(); // missing implementation body
    default void move () { System.out.println("Moving"); }
}
class Cheetah implements Moveable{
    @Override
    public void move () { System.out.println("Moving very fast!"); }
}
class Elephant implements Moveable{}

public class TestAnimal {
    public static void main(String[] args) {
        // cannot new an interface type
        //Moveable m1 = new Moveable();
        Moveable cheetah = new Cheetah();
        cheetah.move();// Moving very fast!
        Moveable elephant = new Elephant();
        elephant.move();// Moving
    }
}
```

```java
package ch10.staticMethods;

interface I{
//    static void m0(); // missing {}
    static int m1(){return 3;}
}
public class TestStaticMethods implements I{
    public static void main(String[] args) {
//        System.out.println(m1());   //  fails to compile
        System.out.println(I.m1()); // 3
    }
}
```

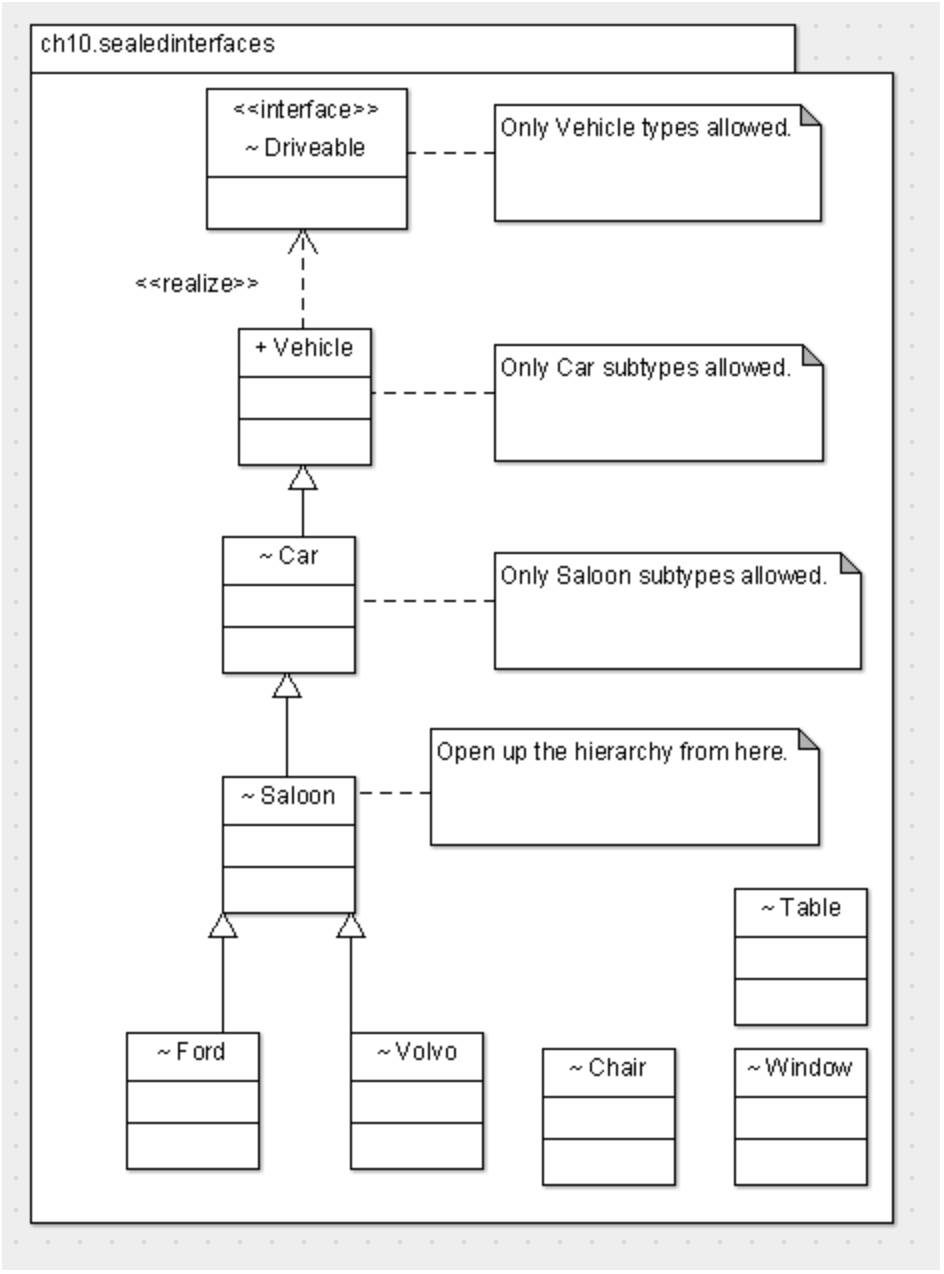
```java
interface A{
    default void foo(){System.out.println("A::foo");}
}
interface B{
    default void foo(){System.out.println("B::foo\n");}
}
public class TestMultipleInheritance implements A, B{
    @Override
    public void foo(){
        System.out.println("TestMultipleInheritance::foo");
        A.super.foo(); // A::foo
//        A.foo();     // does not compile (foo() assumed 'static')
        B.super.foo(); // B::foo
    }
    public static void main(String[] args) {
//        A.super.foo();      // fails to compile
        new TestMultipleInheritance().foo();
    }
}
```

```java
package ch10.privatemethods;

interface InefficientTennis{
    // Lots of code duplication here.
    static void forehand(){
        System.out.println("Move into position");
        System.out.println("Hitting a forehand");
        System.out.println("Move back into ready position");
    }
    default void backhand(){
        System.out.println("Move into position");
        System.out.println("Hitting a backhand");
        System.out.println("Move back into ready position");
    }
    default void smash(){
        System.out.println("Move into position");
        System.out.println("Hitting a smash");
        System.out.println("Move back into ready position");
    }
}
```

```java
interface EfficientTennis{
    // encapsulated within the interface
    private static void hit(String stroke){
        System.out.println("Move into position");
        System.out.println("Hitting a "+stroke);
        System.out.println("Move back into ready position");
    }
    default void backhand(){ hit( stroke: "backhand"); }
    static void forehand(){
//        smash();// static to instance not allowed!
        hit( stroke: "forehand");
    }
    private void smash(){ hit( stroke: "smash"); }
//    void volley(){ hit("volley"); } // abstract by default
}
```

```java
// No abstract methods to implement.
public class SportTest implements EfficientTennis{
    public static void main(String[] args) {
        new SportTest().backhand(); // default method
        EfficientTennis.forehand(); // static method
//        new SportTest().hit("Serve");    // private method
    }
}
```


ch10.sealedinterfaces
<<interface>>
~ Driveable
Only Vehicle types allowed.
<<realize>>
+ Vehicle
Only Car subtypes allowed.
~ Car
Only Saloon subtypes allowed.
Open up the hierarchy from here.
~ Saloon
~ Table
~ Ford
~ Volvo
~ Chair
~ Window

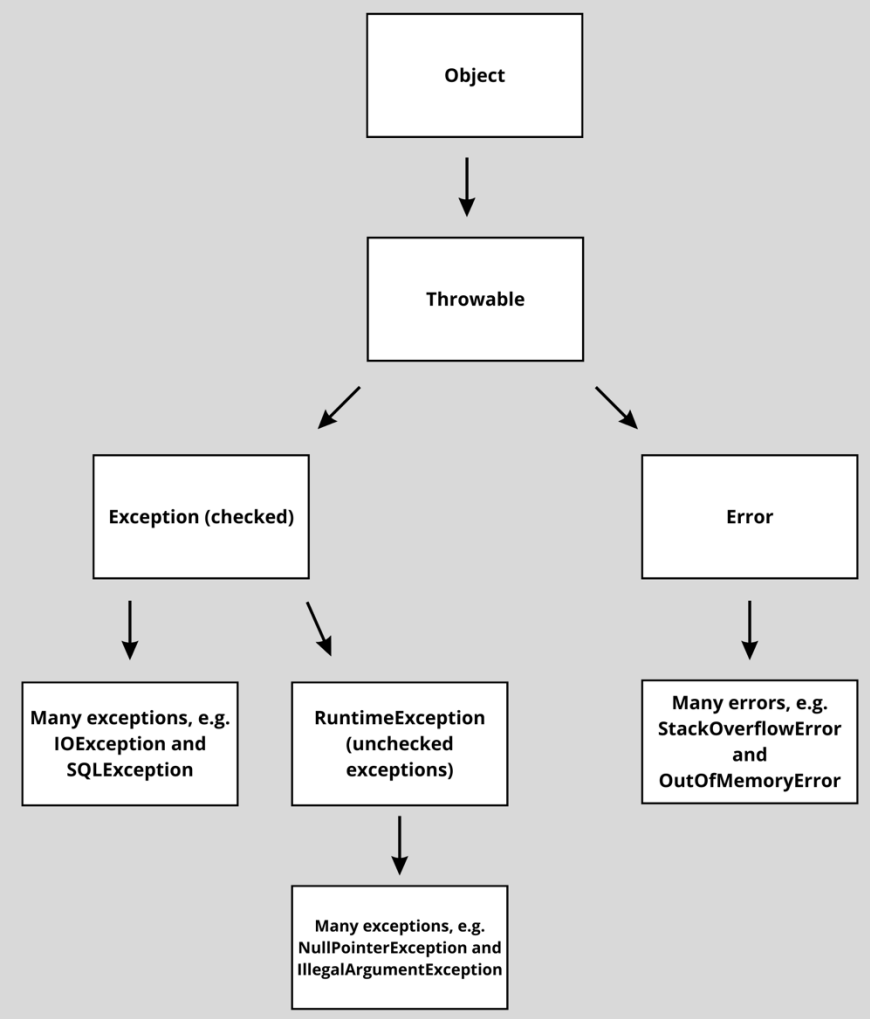
```java
package ch10.sealedinterfaces;

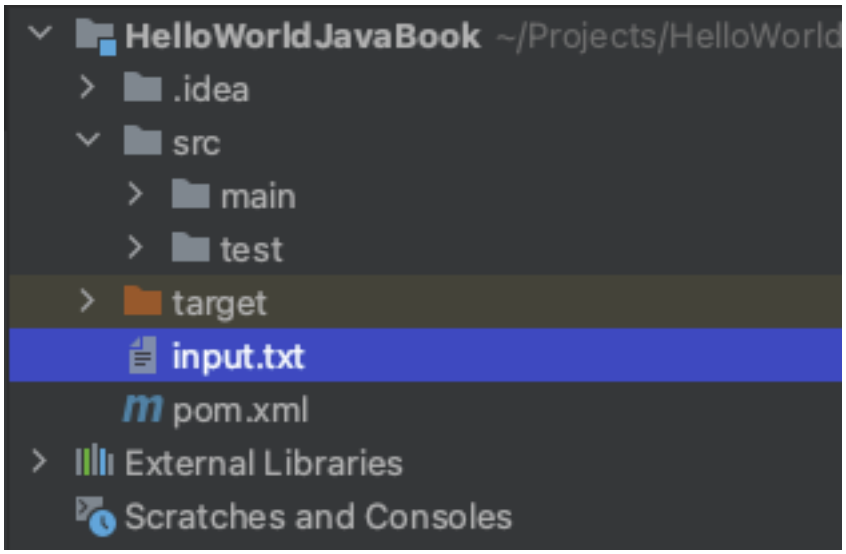
sealed interface Driveable permits Vehicle{ } // scoping interface hierarchy
public sealed class Vehicle implements Driveable permits Car{ }

sealed class Car extends Vehicle permits Saloon {}
non-sealed class Saloon extends Car{}       // opening up hierarchy again
class Volvo extends Saloon{}
class Ford extends Saloon{}

class Window extends Vehicle{}      // Vehicle permits Car only
class Chair extends Car{}           // Car permits Saloon only
class Table implements Driveable{}  // Driveable permits Vehicle only
```

# Chapter 11: Dealing with Exceptions



# Chapter 12: Java Core API

```
22      public static void usingKeyboard(){
23          Scanner sc = new Scanner(System.in);
24          System.out.print("Enter age: ");
25          if(sc.hasNextInt()){ // integer ready to be read
26              int age = sc.nextInt();
27              System.out.println(age);
28          }
29          // closes the underlying stream (System.in) which cannot be re-opened
30  //          sc.close();
31  //          Scanner sc1 = new Scanner(System.in);
32  //          System.out.print("Enter another age: ");
33  //          int age = sc1.nextInt();    // NoSuchElementException
34  //          System.out.println(age);
35      }
```

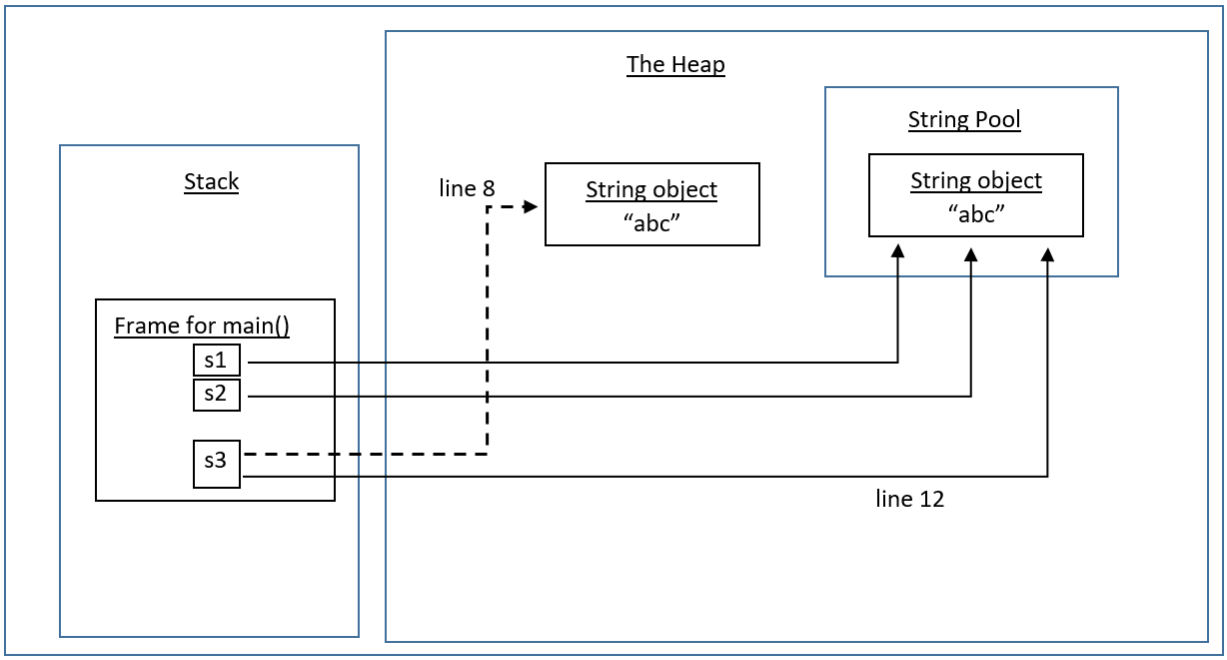
```
14      public static void usingKeyboardExtra(){
15          Scanner sc = new Scanner(System.in);
16          System.out.print("Enter name: ");   // Type in Sean Kennedy
17          System.out.println(sc.next());      // Sean
18      }
```

```
14      public static void usingKeyboardExtra(){
15          Scanner sc = new Scanner(System.in);
16          System.out.print("Enter name: ");
17          System.out.println(sc.nextLine()); // Sean Kennedy
18      }
```

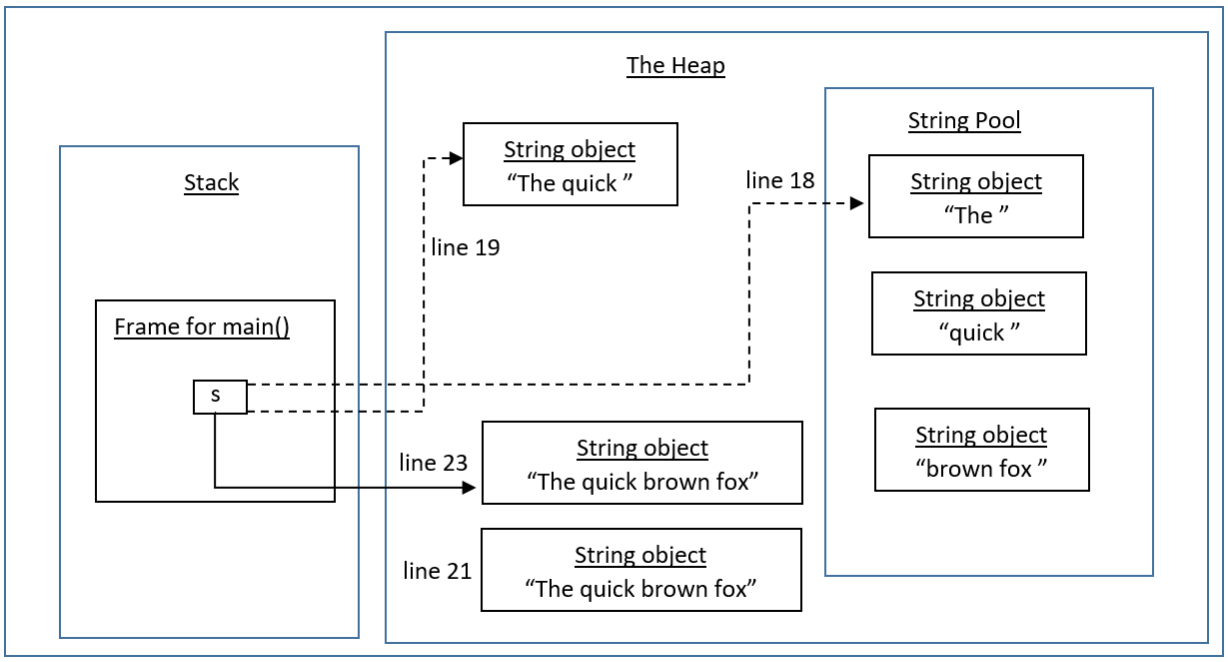
```java
public static void usingFile(){
    // Relative path built from "current working directory", which is
    // obtained from user.dir setting -> System.getProperty("user.dir")
    // Mine was: C:\Users\skennedy\IdeaProjects\JavaFromBeginnerToProfessional
    try (Scanner sc = new Scanner(
                        new File( pathname: "out\\production\\" +
                                "JavaFromBeginnerToProfessional\\ch12\\ages.txt"))) {
        if(sc.hasNextInt()){
            int age = sc.nextInt();
            System.out.println(age);
        }
    } catch (FileNotFoundException fnfe) {
        fnfe.printStackTrace();
    }
}
```

```java
public static void usingString(){
    String input = "Maaike   delim vandelim Putten delim 22";
    Scanner sc = new Scanner(input).useDelimiter( pattern: "\\s*delim\\s*");
    System.out.println(sc.next());
    System.out.println(sc.next());
    System.out.println(sc.next());
    System.out.println(sc.nextInt());
    sc.close();
}
```

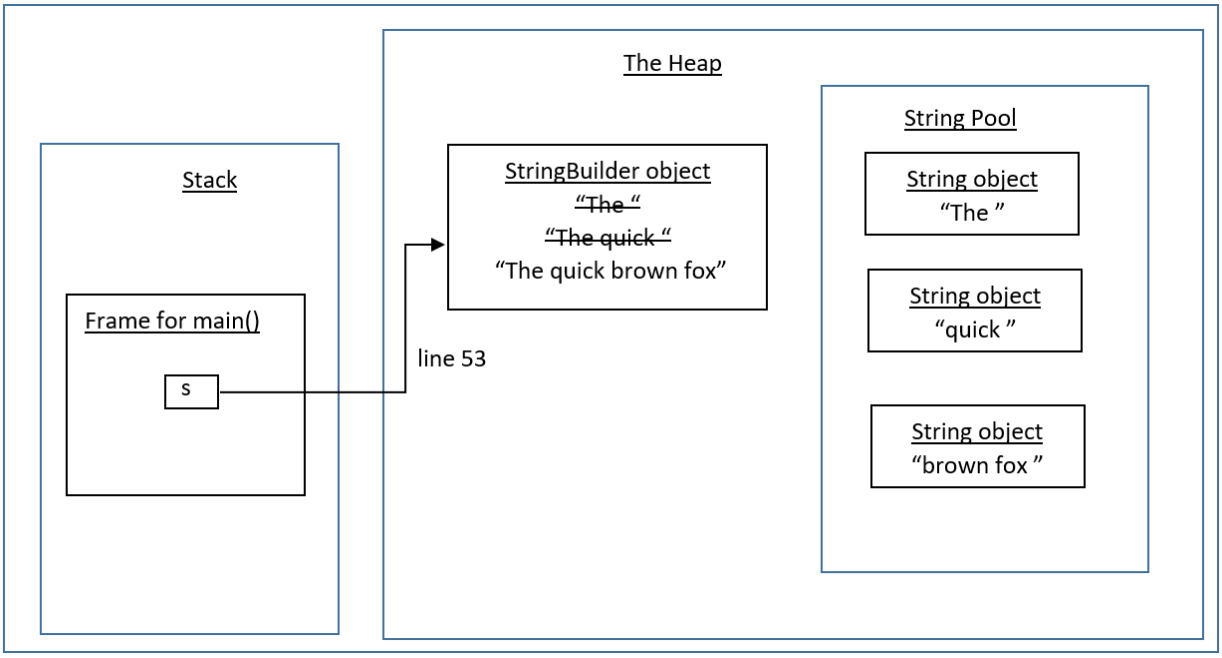
```java
public class StringTest {
    public static void main(String[] args) {
        String s1 = "abc";                      // string pool
        String s2 = "abc";                      // string pool
        System.out.println(s1 == s2);           // true
        String s3 = new String( original: "abc"); // heap
        System.out.println(s1 == s3);           // false
        System.out.println(s1.equals(s2));      // true
        System.out.println(s1.equals(s3));      // true
        s3 = s3.intern();
        System.out.println(s1 == s3);           // true
    }
}
```



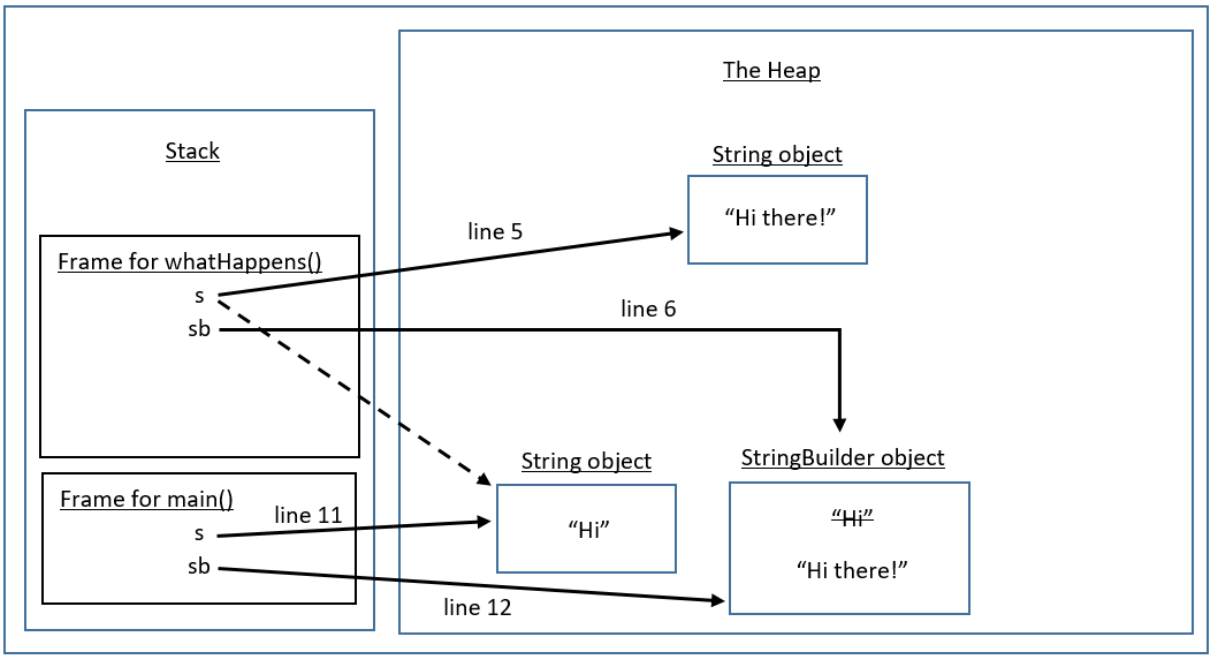

```
17      public static void howManyObjectsString(){
18          String s = "The ";
19          s += "quick ";           // s = s + "quick "
20          System.out.println(s);   // The quick
21          s.concat( str: "brown fox");// lost!
22          System.out.println(s);   // The quick
23          s = s.concat( str: "brown fox");
24          System.out.println(s);   // The quick brown fox
25      }
```




```
48      public static void howManyObjectsSB(){
49          StringBuilder sb = new StringBuilder("The ");
50          sb.append("quick ");
51          System.out.println(sb);   // The quick
52          sb.append("brown fox");
53          System.out.println(sb);  // The quick brown fox
54      }
```




```
3  public class StringVersusStringBuilder {
4  @    public static void whatHappens(String s, StringBuilder sb){
5           s = s.concat( str: " there!");
6           sb.append(" there!");
7           System.out.println("whatHappens: "+s);  // Hi there!
8           System.out.println("whatHappens: "+sb); // Hi there!
9       }
10      public static void main(String[] args) {
11          String s = "Hi";
12          StringBuilder sb = new StringBuilder("Hi");
13          whatHappens(s, sb);
14          System.out.println("main: "+s);  // Hi
15          System.out.println("main: "+sb); // Hi there!
16      }
17  }
```


The Heap
Stack
String object
"Hi there!"
line 5
Frame for whatHappens()
s
sb
line 6
String object
StringBuilder object
Frame for main()
line 11
s
sb
"Hi"
~~"Hi"~~
"Hi there!"
line 12

```java
import java.util.ArrayList;
import java.util.List;

final class Farm { // cannot subclass this class and all methods are final
    // private final instance variables
    private final String name; // String is immutable
    private final int numAnimals;
    private final List<String> animals;// mutable

    // private constructor
    private Farm(final String name, final int numAnimals, final List<String> animals){
        this.name           = name;
        this.numAnimals     = numAnimals;
//        this.animals      = new ArrayList<String>(animals); // create a new ArrayList
        this.animals        = animals; // breaking encapsulation!
    }
    // factory method to create a Farm
    public static Farm createNewInstance(String name, int numAnimals,
                                          List<String> animals){
        return new Farm(name, numAnimals, animals);
    }
    // no 'set' methods, only 'get' methods
    public String getName() { return name; }
    public int getNumAnimals() { return numAnimals; }
    public List<String> getAnimals(){
//        return new ArrayList<String>(animals);  // return a new object
        return animals;  // breaking encapsulation!
    }
    @Override
    public String toString() {
        return "Farm{" + "name=" + name + ", numAnimals=" +
                numAnimals + ", animals=" + animals + '}';
    }
}
```

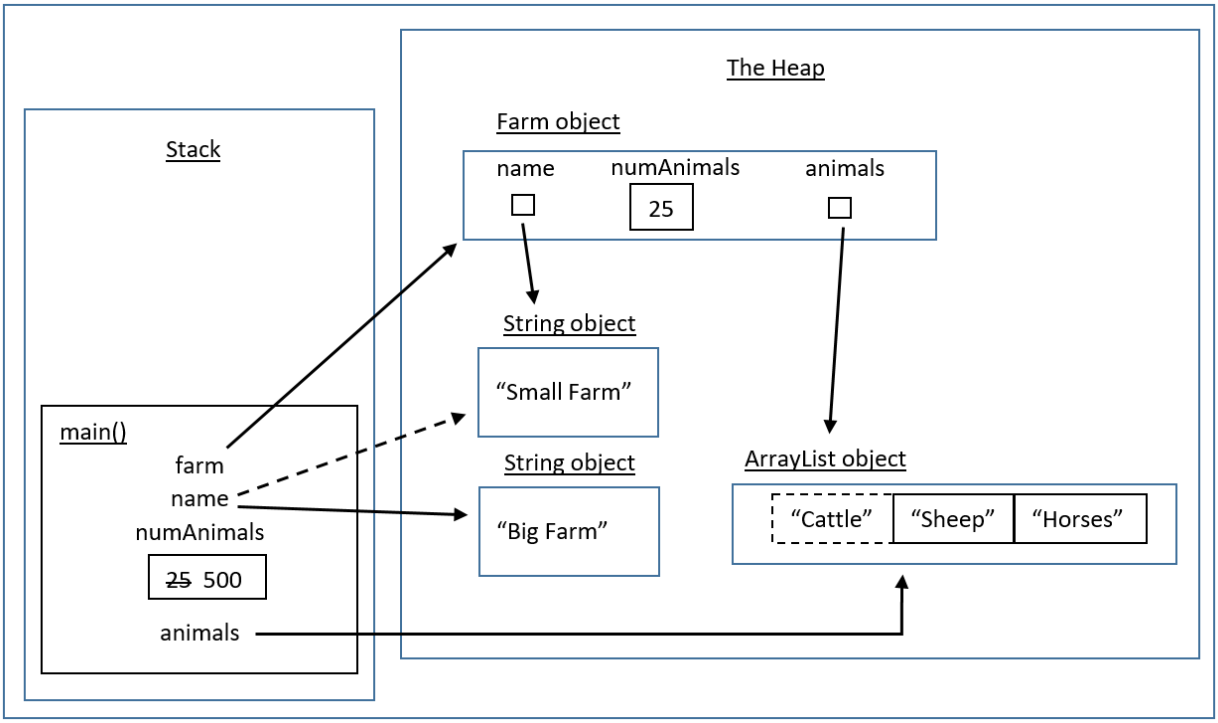
```java
public class TestImmutable {
    public static void main(String[] args) {
        List<String> animals = new ArrayList<>();
        animals.add("Cattle");

        Farm farm = Farm.createNewInstance( name: "Small Farm", numAnimals: 25, animals);
        System.out.println("Created: "+farm); // Created: Farm{name=Small Farm, numAnimals=25, animals=[Cattle]}

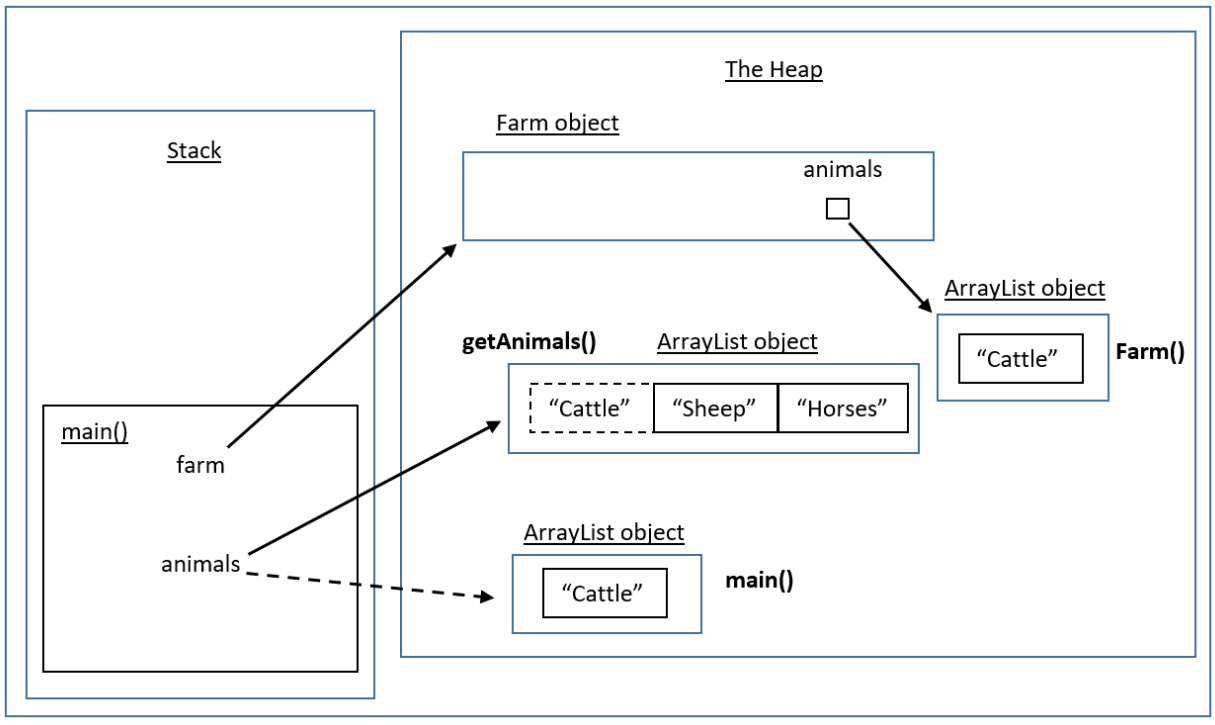
        // Get the instance variables
        String name    = farm.getName();
        int numAnimals = farm.getNumAnimals();
        animals        = farm.getAnimals();
        System.out.println("Retrieved: "+name+" "
                +" "+numAnimals+ " "+animals); // Retrieved: Small Farm  25 [Cattle]

        // change what I got back - any affect on the "farm" immutable object?
        name = "Big Farm";// Strings are immutable so new objects are created in the background => OK
        numAnimals = 500; // simple primitive i.e. value is just copied back
        animals.add("Sheep");animals.add("Horses");  // safe or unsafe ?

        // Any change?: Farm{name=Small Farm, numAnimals=25, animals=[Cattle, Sheep, Horses]}
        System.out.println("Any change?: "+farm);
    }
}
```



```java
    // private constructor
    private Farm(final String name, final int numAnimals, final List<String> animals){
        this.name           = name;
        this.numAnimals     = numAnimals;
        this.animals      = new ArrayList<String>(animals); // create a new ArrayList
//        this.animals        = animals; // breaking encapsulation!
    }
    // factory method to create a Farm
    public static Farm createNewInstance(String name, int numAnimals,
                                          List<String> animals){
        return new Farm(name, numAnimals, animals);
    }
    // no 'set' methods, only 'get' methods
    public String getName() { return name; }
    public int getNumAnimals() { return numAnimals; }
    public List<String> getAnimals(){
        return new ArrayList<String>(animals);  // return a new object
//        return animals;  // breaking encapsulation!
    }
```



```java
public static void listProperties() {
    List<String> list = new ArrayList<>();
    list.add("J");
    list.add("A");
    list.add("V");
    list.add("A");
    // insertion order maintained, duplicates allowed
    System.out.println(list); // [J, A, V, A]
    list.add( index: 1, element: "O");
    list.add( index: 3, element: "O");
    // precise control of where elements are inserted
    System.out.println(list); // [J, O, A, O, V, A]
}
```

```java
List<String> list = new ArrayList<>();
list.add("Joe"); list.add("Mary"); list.add("Joe");
System.out.println(list.contains("Mary"));  // true
System.out.println(list.get(0));            // Joe
System.out.println(list.indexOf(2));        // -1, no such Object
System.out.println(list.indexOf("Joe"));    // 0, first occurrence
System.out.println(list.remove( o: "Joe")); // true, first occurrence
System.out.println(list);                   // [Mary, Joe]
list.remove( index: 0);
System.out.println(list);                   // [Joe]
list.set(0, "Paul");
System.out.println(list);                   // [Paul]
```

```java
LocalDate nowDate = LocalDate.now(); // get current date from system clock
LocalTime nowTime = LocalTime.now();
LocalDateTime nowDateTime2 = LocalDateTime.now();   // one way
LocalDateTime nowDateTime = LocalDateTime.of(nowDate, nowTime);// another way
System.out.println(nowDateTime);// 2023-07-09T13:28:00.322907600

// Setting St. Patricks Day, 2025
LocalDate ld1 = LocalDate.of( year: 2025,  month: 3,  dayOfMonth: 17);// one way
LocalDate ld2 = LocalDate.parse( text: "2025-03-17");// another way
System.out.println(ld2.getDayOfWeek());// MONDAY
LocalDate ld3 = ld2.withMonth(5);
System.out.println(ld3);// 2025-05-17
LocalDate ld4 = ld3.plusYears( yearsToAdd: 1);
System.out.println(ld4);// 2026-05-17
LocalDate ld5 = ld4.minusDays( daysToSubtract: 5);
System.out.println(ld5);// 2026-05-12
LocalDateTime ldt1 = ld5.atTime( hour: 13,  minute: 45,  second: 10);
System.out.println(ldt1);// 2026-05-12T13:45:10
```

```java
LocalDateTime flightDepTime  = LocalDateTime.of( year: 2023,
                                                 Month.NOVEMBER,
                                                 dayOfMonth: 24,
                                                 hour: 13,
                                                 minute: 00);
ZonedDateTime flightDepTimeZ = flightDepTime.atZone(ZoneId.of( zoneId: "Europe/Dublin"));
System.out.println(flightDepTimeZ);// 2023-11-24T13:00Z[Europe/Dublin]

ZonedDateTime arrivalTimeZ  =
        flightDepTimeZ.withZoneSameInstant(ZoneId.of( zoneId: "Europe/Paris"))
                .plusHours(1)
                .plusMinutes(45);

System.out.println(arrivalTimeZ); // 2023-11-24T15:45+01:00[Europe/Paris]
System.out.println(arrivalTimeZ.getHour()+":"+arrivalTimeZ.getMinute()); // 15:45
```

```java
// Period
LocalDate startDate = LocalDate.of( year: 1861,  month: 4,  dayOfMonth: 12);
LocalDate endDate   = LocalDate.of( year: 1865,  month: 4,  dayOfMonth: 9);
Period howLongP     = Period.between(startDate,endDate);
System.out.println(howLongP);            // P3Y11M28D, weeks not represented
System.out.println(howLongP.getYears()); // 3
System.out.println(howLongP.getMonths());// 11
System.out.println(howLongP.getDays());  // 28

// Duration
LocalTime lt1 = LocalTime.of( hour: 12, minute: 0,  second: 20);
LocalTime lt2 = LocalTime.of( hour: 14, minute: 45,  second: 40);
Duration howLongD2 = Duration.between(lt1, lt2);
System.out.println(howLongD2);// PT2H45M20S
```

```java
LocalDate date = LocalDate.now();
DateTimeFormatter isoDate = DateTimeFormatter.ISO_DATE;
System.out.println(date.format(isoDate));       // 2023-07-10
DateTimeFormatter fullDateStyle =
        DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);
System.out.println(date.format(fullDateStyle));  // Monday 10 July 2023

LocalTime time = LocalTime.now();
DateTimeFormatter isoTime = DateTimeFormatter.ISO_TIME;
System.out.println(time.format(isoTime));           // 15:47:13.3961956
DateTimeFormatter shortTimeStyle =
        DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
System.out.println(time.format(shortTimeStyle));    // 15:47
```

```java
ZonedDateTime zdt = ZonedDateTime.now();
System.out.println(zdt);      // 2023-07-11T09:05:50.792542600+01:00[Europe/Dublin]
DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("yy-MMM-dd E D");
System.out.println(zdt.format(dateFormatter));// 23-Jul-11 Tue 192

DateTimeFormatter timeFormatter1 = DateTimeFormatter.ofPattern("hh:mm:ss a z G");
System.out.println(zdt.format(timeFormatter1));// 09:05:50 a.m. IST AD

// how to insert text
DateTimeFormatter dateFormatter2 =
        DateTimeFormatter.ofPattern("'Year: 'yyyy'. Month: 'MMMM'. Day: 'dd'.'");
System.out.println(zdt.format(dateFormatter2));// Year: 2023. Month: July. Day: 11.

// parse
String dateTimeString = "2023-07-10 22:10"; // last night
DateTimeFormatter timeFormatter2 = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
LocalDateTime ldt = LocalDateTime.parse(dateTimeString, timeFormatter2);
System.out.println(ldt);    // 2023-07-10T22:10
```

# Chapter 13: Generics and Collections

***No-images…***

# Chapter 14: Lambda Expressions

```java
@FunctionalInterface // a functional interface
interface I{
    // one abstract method
    void m();
}
public class BasicLambdas {
    public static void main(String[] args) {
        // define the lambda
        I lambda1 = () -> {
            System.out.println("First lambda!");
        };
        // execute the lambda
        lambda1.m(); // First lambda!

        I lambda2 = () -> System.out.println("Second lambda!");
        lambda2.m(); // Second lambda!

    }
}
```

```java
// Functional Interface
@FunctionalInterface
interface Evaluate<T> {
    boolean check(T t);
}
public class TestPredicate {
    public static void main(String[] args) {
        Evaluate<Integer> isItPositive = (Integer n) -> {return n > 0;};
        System.out.println(isItPositive.check( t: -1));//false
        System.out.println(isItPositive.check( t: +1));//true

        Evaluate<String> isMale = s -> s.startsWith("Mr.");
        System.out.println(isMale.check( t: "Mr. Sean Kennedy"));//true
        System.out.println(isMale.check( t: "Ms. Maaike van Putten"));//false
    }
}
```

```java
public class LambdaEffectivelyFinal {
    public static void main(String[] args) {
        ArrayList<String> people = new ArrayList<>();
        people.add ("Mr. John Bloggs");people.add ("Ms. Ann Bloggs");
        people.add ("Mr. Mike Bloggs");people.add ("Ms. Mary Bloggs");

        String title="Mr."; // final or effectively final
        int y = 0;
        y++; // no issue, 'y' is not used in lambda

        // Lambdas take a snapshot/picture of local variables; these local
        // variables MUST NOT change. Only setting up lambda here.
        Predicate<String> lambda = str -> {
            //title = "Miss;
            return str.startsWith(title); // "Mr."
        };

        // If 'title' was allowed to change, then the method and the lambda would
        // have 2 different views of 'title'!
        //title = "Ms";
        filterData(people, lambda);// lambda views 'title' as "Mr."

        //title = "Ms";
        filterData(people, lambda);// lambda views 'title' as "Mr."
    }
    public static void filterData(List<String> list, Predicate<String> lambda){
        list.removeIf(lambda); // only executing lambda here!
    }
}
```

```java
    public void predicate() {
        // Predicate<T> is a functional interface i.e. one abstract method:
        //      boolean test(T t);
        Predicate<String> pStr = s -> s.contains("City");
        System.out.println(pStr.test( t: "Vatican City"));//true

        // BiPredicate<T, U> is a functional interface i.e. one abstract method:
        //      boolean test(T t, U u);
        BiPredicate<String, Integer> checkLength = (str, len) -> str.length() == len;
        System.out.println(checkLength.test( t: "Vatican City",  u: 8));//false (length is 12)

    }
```

```java
// Supplier<T> is a functional interface i.e. one abstract method:
//      T get()
Supplier<StringBuilder> supSB = () -> new StringBuilder();
System.out.println(supSB.get().append("SK"));// SK

Supplier<LocalTime> supTime = () -> LocalTime.now();
System.out.println(supTime.get());  // 11:00:50.769271700

Supplier<Double> sRandom = () -> Math.random();
System.out.println(sRandom.get());  // 0.13482391499981883
```

```java
// Consumer<T> is a functional interface i.e. one abstract method:
//      void accept(T t)
Consumer<String> printC = s -> System.out.println(s);// lambda
printC.accept( t: "To be or not to be, that is the question");

List<String> names = new ArrayList<>();
names.add("Maaike");names.add("Sean");
names.forEach(printC);  // Maaike, Sean
```

```java
Map<String,String> mapCapitalCities = new HashMap<>();
// BiConsumer<T, U> is a functional interface i.e. one abstract method:
//      void accept(T t, U u)
// Note: Object put(k,v) - return value ignored.
//       This Consumer has a side-effect.
BiConsumer<String, String> biCon = (key, value) ->
                                   mapCapitalCities.put(key, value);
biCon.accept( t: "Dublin",  u: "Ireland");
biCon.accept( t: "The Hague",  u: "Holland");
System.out.println(mapCapitalCities);// {Dublin=Ireland, The Hague=Holland}

BiConsumer<String, String> mapPrint = (key, value) ->
                                          System.out.println(key +
                                                  " is the capital of: "+value);
mapCapitalCities.forEach(mapPrint); // Dublin is the capital of: Ireland
                                    // The Hague is the capital of: Holland
```

```java
// Function<T, R> is a functional interface i.e. one abstract method:
//      R apply(T t)
Function<String, Integer> fn2 = s -> s.length();
System.out.println("Function: " + fn2.apply( t: "London"));// 6

// BiFunction<T, U, R> is a functional interface i.e. one abstract method:
//      R apply(T t, U u)
BiFunction<String, String, Integer> biFn =
        (s1, s2) -> s1.length() + s2.length();
System.out.println("BiFunction: " +
        biFn.apply( t: "William",  u: "Shakespeare"));// 18

BiFunction<String, String, String> biFn2 =
        (s1, s2) -> s1.concat(s2);
System.out.println("BiFunction: " +
        biFn2.apply( t: "William ",  u: "Shakespeare"));// William Shakespeare
```

```java
   // UnaryOperator<T> extends Function<T, T>
   //      T apply(T t)
   UnaryOperator<String> unaryOp = name -> "My name is "+name;
   System.out.println("UnaryOperator: " +
           unaryOp.apply( t: "Sean"));// My name is Sean
```

```java
// BinaryOperator<T> extends BiFunction<T, T, T>
//      T apply(T t1, T t2)
BinaryOperator<String> binaryOp   = (s1, s2) -> s1.concat(s2);
System.out.println("BinaryOperator: " +
       binaryOp.apply( t: "William ",  u: "Shakespeare"));// William Shakespeare
```

```
21          String name = "Mr. Joe Bloggs";
22          // Supplier<T>
23          //      T get()
24          Supplier<String> lowerL   = () -> name.toLowerCase();   // lambda
25          Supplier<String> lowerMR  = name::toLowerCase;          // method reference
26
27          // No need to say which instance to call it on - the supplier is bound to name
28          System.out.println(lowerL.get()); // mr. joe bloggs
29          System.out.println(lowerMR.get());// mr. joe bloggs
30
31          // Predicate<T>
32          //    boolean test(T t)
33          // Even though startsWith is overloaded, boolean startsWith(String) and
34          // boolean startsWith(String, int), because we are creating a Predicate which
35          // has a functional method of test(T t), the startsWith(String) is used.
36          // This is where "context" is important.
37          Predicate<String> titleL  = (title) -> name.startsWith(title);
38          Predicate<String> titleMR = name::startsWith;
39
40          System.out.println(titleL.test( t: "Mr.")); // true
41          System.out.println(titleMR.test( t: "Ms."));// false
```

```
44      public static void unboundMethodReferences(){
45          //   Function<T, R>
46          //      R apply(T)
47          //          String apply(String)
48          Function<String, String> upperL  = s -> s.toUpperCase();
49          Function<String, String> upperMR = String::toUpperCase;
50          // The function is unbound, so you need to specify which instance to call it on
51          System.out.println(upperL.apply( t: "sean"));   // SEAN
52          System.out.println(upperMR.apply( t: "sean"));  // SEAN
53
54          //   Function<T, U, R>
55          //      R apply(T t, U u)
56          //          String apply(String, String)
57          BiFunction<String, String, String> concatL   = (s1, s2) -> s1.concat(s2);
58          BiFunction<String, String, String> concatMR  = String::concat;
59          System.out.println(concatL.apply( t: "Sean ",  u: "Kennedy"));// Sean Kennedy
60
61          // 1st parameter is used for executing the instance method
62          // "Sean ".concat("Kennedy")
63          System.out.println(concatMR.apply( t: "Sean ",  u: "Kennedy"));// Sean Kennedy
64      }
```

```java
//  Static method references are considered UNBOUND also. An example static method
//  is Collections.sort(List)
//  Consumer<T>
//      void accept(T t)
//          void accept(List<Integer>)
//  NB: Consumer takes one parameter => sort(List) is used, as opposed to sort(List, Comparator)
Consumer<List<Integer>> sortL  = list -> Collections.sort(list);
Consumer<List<Integer>> sortMR = Collections::sort;

List<Integer> listOfNumbers = Arrays.asList(2,1,5,4,9);
sortL.accept(listOfNumbers);// execution
System.out.println(listOfNumbers);  // [1, 2, 4, 5, 9]

listOfNumbers = Arrays.asList(8,12,4,3,7);
sortMR.accept(listOfNumbers);// execution
System.out.println(listOfNumbers);  // [3, 4, 7, 8, 12]
```

```java
// Supplier<T>
//      T get()
Supplier<StringBuilder> sbL   = () -> new StringBuilder();  // lambda
Supplier<StringBuilder> sbMR  = StringBuilder::new;         // method reference
StringBuilder sb1 = sbL.get(); sb1.append("lambda version"); System.out.println(sb1);
StringBuilder sb2 = sbMR.get(); sb2.append("method reference version"); System.out.println(sb2);

//  Function<T, R>
//      R apply(T)
//          List<String> apply(Integer)
//  ArrayList(int initialCapacity)
Function<Integer, List<String>> alL  = x -> new ArrayList(x);
Function<Integer, List<String>> alMR = ArrayList::new;
List<String> ls1 = alL.apply( t: 100);  // initial capacity 100
ls1.add("21");
System.out.println(ls1);//[21]
List<String> ls2 = alMR.apply( t: 200);  // initial capacity 200
ls2.add("88");
System.out.println(ls2);//[88]
```

```java
class Person{
    public static Integer howMany(Person... people){
        return people.length;
    }
}
public class MethodRefsAndContext {
    public static void main(String[] args) {
        // No Person being passed in => Supplier
        Supplier<Integer> lambda1 = () -> Person.howMany();
        Supplier<Integer> mr1     = Person::howMany;
        System.out.println(lambda1.get());  // 0
        System.out.println(mr1.get());      // 0

        // One Person to be passed in => Function
        Function<Person, Integer> lambda2 = person -> Person.howMany(person);
        Function<Person, Integer> mr2     = Person::howMany;
        System.out.println(lambda2.apply(new Person()));  // 1
        System.out.println(mr2.apply(new Person()));      // 1

        // Two Person's to be passed in => BiFunction
        BiFunction<Person, Person, Integer> lambda3 = (p1, p2) -> Person.howMany(p1, p2);
        BiFunction<Person, Person, Integer> mr3     = Person::howMany;
        System.out.println(lambda3.apply(new Person(), new Person()));  // 2
        System.out.println(mr3.apply(new Person(), new Person()));      // 2
    }
}
```

# Chapter 15: Streams – Fundamentals

```
7           var temps = Arrays.asList(98.4, 100.2, 87.9, 102.8);
8           long count = temps.stream()
9                   .peek(System.out::println) // 98.4, 100.2, 87.9, 102.8
10                  .filter(temp -> temp > 100)
11                  .peek(System.out::println) //       100.2,       102.8
12                  .count();
13          System.out.println(count);  // 2
```

```
20          List<String> names = Arrays.asList("April", "Ben", "Charlie",
21                          "David", "Benildus", "Christian");
22          names.stream()
23                  .peek(System.out::println)
24                  .filter(s -> {
25                      System.out.println("filter1 : "+s);
26                      return s.startsWith("B") || s.startsWith("C"); } )
27                  .filter(s -> {
28                      System.out.println("filter2 : "+s);
29                      return s.length() > 3; } )
30                  .limit( maxSize: 1) // intermediate operation  Stream<T> limit(long)
31                  .forEach(System.out::println);  // terminal operation
```

```
34  /* Output:
35  April                   - peek
36  filter1 : April         - filter1 removes April
37  Ben                     - peek
38  filter1 : Ben           - filter1 passes Ben on
39  filter2 : Ben           - filter2 removes Ben
40  Charlie                 - peek
41  filter1 : Charlie       - filter1 passes Charlie on
42  filter2 : Charlie       - filter2 passes Charlie on
43  Charlie                 - forEach()
44
45  Note: limit(1) means David, Benildus or Christian are not
46        processed at all i.e. none of them appear in the output
47        via "peek()"
48  */
```

```
80          Double[] numbers = {1.1, 2.2, 3.3};
81          Stream<Double> stream1 = Arrays.stream(numbers);
82          long n = stream1.count();
83          System.out.println("Number of elements: "+n);// 3
```

```
58  List<String> animalList = Arrays.asList("cat", "dog", "sheep");
59  // using stream() which is a default method in Collection interface
60  Stream<String> streamAnimals = animalList.stream();
61  System.out.println("Number of elements: "+streamAnimals.count()); // 3
62
63  // stream() is a default method in the Collection interface and therefore
64  // is inherited by all classes that implement Collection. Map is NOT one
65  // of those i.e. Map is not a Collection. To bridge between the two, we
66  // use the Map method entrySet() to return a Set view of the Map (Set
67  // IS-A Collection).
68  Map<String, Integer> namesToAges = new HashMap<>();
69  namesToAges.put("Mike", 22);namesToAges.put("Mary", 24);namesToAges.put("Alice", 31);
70  System.out.println("Number of entries: "+
71          namesToAges
72              .entrySet() // get a Set (i.e. Collection) view of the Map
73              .stream()   // stream() is a default method in Collection
74              .count());  // 3
```

```
15      String[] cities = {"Dublin", "Berlin", "Paris"};
16      Stream<String> citiesStream = Stream.of(cities);
17      System.out.println(citiesStream.count()); // 3
18
19      Stream<Integer> streamI = Stream.of( ...values: 1,2,3);
20      System.out.println(streamI.count()); // 3
21
22      Stream<String> streamS = Stream.of( ...values: "a", "b", "c", "d");
23      System.out.println(streamS.count()); // 4
24
25      Stream<Dog> streamD = Stream.of(new Dog());
26      System.out.println(streamD.count()); // 1
```

```java
// infinite stream of random unordered numbers
// between 0..9 inclusive
//    Stream<T> generate(Supplier<T> s)
//      Supplier is a functional interface:
//          T get()
Stream<Integer> infStream = Stream.generate(() -> {
    return (int) (Math.random() * 10);
});
// keeps going until I kill it.
infStream.forEach(System.out::println);
```

```java
// infinite stream of ordered numbers
//    2, 4, 6, 8, 10, 12 etc...
// iterate(T seed, UnaryOperator<T> fn)
//   UnaryOperator is-a Function<T, T>
//     T apply(T t)
Stream<Integer> infStream = Stream.iterate( seed: 2, n -> n + 2);

// keeps going until I kill it.
infStream.forEach(System.out::println);
```

```java
// finite stream of ordered numbers
// 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
Stream
        .iterate( seed: 2,          // seed
                n -> n <= 20,    // Predicate
                n -> n + 2)      // UnaryOperator
        .forEach(System.out::println);
```

```
36          // finite stream of ordered numbers
37          // 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
38          Stream
39              .iterate( seed: 2, n -> n + 2)
40              // limit() is a short-circuiting stateful
41              // intermediate operation
42              .limit( maxSize: 10)
43              // forEach(Consumer) is a terminal operation
44              .forEach(System.out::println);
```

```
152         long count = Stream.of( ...values: "dog", "cat")
153                           .count();
154         System.out.println(count); // 2
```

```
163     // Optional<T> min(Comparator)
164     // Optional<T> max(Comparator)
165     Optional<String> min = Stream.of( ...values: "deer", "horse", "pig")
166                             .min((s1, s2) -> s1.length()-s2.length());
167     min.ifPresent(System.out::println);// pig
168
169     Optional<Integer> max = Stream.of( ...values: 4,6,2,12,9)
170                             .max((i1, i2) -> i1-i2);
171     max.ifPresent(System.out::println);// 12
172
173     Optional<Object> noMin = Stream.empty().min((x1, x2) -> 0);
174     System.out.println(noMin.isEmpty());// true
175     System.out.println(noMin.isPresent());// false
```

```java
    // Optional<T> findAny()
    // Optional<T> findFirst()
    // These are terminal operations but not reductions
    // as they can return without processing all
    // the elements in the stream. Reductions reduce the
    // entire stream into one value.
    Optional<String> any = Stream.of( ...values: "John", "Paul")
                        .findAny();
    any.ifPresent(System.out::println);// John (usually)

    Optional<String> first = Stream.of( ...values: "John", "Paul")
                        .findFirst();
    first.ifPresent(System.out::println);// John
```

```java
  // boolean anyMatch(Predicate)
  // boolean allMatch(Predicate)
  // boolean noneMatch(Predicate)
  List<String> names = Arrays.asList("Alan", "Brian", "Colin");
  Predicate<String> pred = name -> name.startsWith("A");
  System.out.println(names.stream().anyMatch(pred)); // true ("Alan")
  System.out.println(names.stream().allMatch(pred)); // false ("Brian")
  System.out.println(names.stream().noneMatch(pred));// false ("Alan")
```

```java
Stream<String> infStr = Stream.generate(() -> "abc");
Predicate<String> startsWithA = s -> s.startsWith("a");
Predicate<String> startsWithB = s -> s.startsWith("b");
System.out.println(infStr.anyMatch(startsWithA));   // true, short-circuit
System.out.println(infStr.anyMatch(startsWithB));   // forever...

System.out.println(infStr.noneMatch(startsWithA));  // false
System.out.println(infStr.noneMatch(startsWithB));  // forever...

System.out.println(infStr.allMatch(startsWithA));   // forever...
System.out.println(infStr.allMatch(startsWithB));   // false, short-circuit
```

```java
  // void forEach(Consumer)
  // As there is no return value, forEach() is not a reduction.
  // As the return type is 'void', if you want something to
  // happen, it has to happen inside the Consumer (side-effect).
  Stream<String> names = Stream.of( ...values: "Cathy", "Pauline", "Zoe");
  names.forEach(System.out::print);//CathyPaulineZoe
```

```java
        // T reduce(T identity, BinaryOperator<T> accumulator)
        //      BinaryOperator<T> functional method:
        //          T apply(T, T);
        String name = Stream.of( ...values: "s", "e", "a", "n")
                        .reduce( identity: "", (s1, s2) -> s1 + s2);
        System.out.println(name);// sean

        String name2 = Stream.of( ...values: "s", "e", "a", "n")
                        .filter(s -> s.length()>2)
                        .reduce( identity: "nothing", (s1, s2) -> s1 + s2);
        System.out.println(name2);// nothing

        Integer product = Stream.of( ...values: 2,3,4)
                        .reduce( identity: 1, (n1, n2) -> n1 * n2);
        System.out.println(product);// 24
```

```java
    // <U> U reduce (U identity,
    //               BiFunction accumulator,
    //               BinaryOperator combiner)
    Stream<String> stream = Stream.of( ...values: "car", "bus", "train", "aeroplane");
    int length = stream.reduce( identity: 0,  // identity
                                (n, str) -> n + str.length(), // n is Integer
                                (n1, n2) -> n1 + n2); // both are Integers
    System.out.println(length);// 20
```

```java
public static void doCollect1(){

    // StringBuilder collect(Supplier<StringBuilder> supplier,
    //               BiConsumer<StringBuilder,String> accumulator
    //               BiConsumer<StringBuilder,StringBuilder> combiner)
    StringBuilder word = Stream.of( ...values: "ad", "jud", "i", "cate")
            .collect(() -> new StringBuilder(),         // StringBuilder::new
                    (sb, str) -> sb.append(str),        // StringBuilder::append
                    (sb1, sb2) -> sb1.append(sb2));     // StringBuilder::append
    System.out.println(word);// adjudicate
```

```java
        String s = Stream.of( ...values: "cake", "biscuits", "apple tart")
                        .collect(Collectors.joining( delimiter: ", "));
        System.out.println(s);  // cake, biscuits, apple tart
```

```java
Double avg = Stream.of( ...values: "cake", "biscuits", "apple tart")
                // averagingInt(ToIntFunction) functional method is:
                //      int applyAsInt(T value);
        .collect(Collectors.averagingInt(s -> s.length())); // String::length
System.out.println(avg);  // 7.333333333333333
```

```java
class Car{
    private String brand;
    private int year;
    public Car(String brand, int year) {
        this.brand = brand;
        this.year  = year;
    }
    public String getBrand() {
        return brand;
    }
    // other methods omitted
}
```

```java
var cars = new ArrayList<Car>();
cars.add(new Car( brand: "Tesla",  year: 2021));
cars.add(new Car( brand: "Ford",  year: 2022));
cars.add(new Car( brand: "Audi",  year: 2018));
List<String> list = cars.stream()
        .map(car -> car.getBrand())// Car::getBrand
        .collect(Collectors.toList());
System.out.println(list);// [Tesla, Ford, Audi]
```

```java
var cars = new ArrayList<Car>();
cars.add(new Car( brand: "Tesla",  year: 2021));
cars.add(new Car( brand: "Ford",  year: 2022));
cars.add(new Car( brand: "Audi",  year: 2018));
Set<String> treeSet = cars.stream()
        .map(car -> car.getBrand())// Car::getBrand
        .collect(Collectors.toCollection(TreeSet::new));
System.out.println(treeSet);// [Audi, Ford, Tesla]
```

```
175         // We want a map: dessert name -> number of characters in dessert name
176         // Output:
177         //   {biscuits=8, cake=4, apple tart=10}
178         Map<String, Integer> map =
179                 Stream.of( ...values: "cake", "biscuits", "apple tart")
180                 .collect(
181 //                      Collectors.toMap(s -> s,          // lambda key
182 //                                       s -> s.length()) // lambda value
183                         Collectors.toMap(String::toString,  // Function for the key
184                                          String::length)    // Function for the value
185                 );
186         System.out.println(map);
```

```
176         // We want a map: number of characters in dessert name -> dessert name.
177         // However, 2 of the desserts have the same length (cake and tart).
178         // As length is our key and we can't have duplicate keys, this leads to an
179         // exception as Java does not know what to do...
180         //    IllegalStateException: Duplicate key 4 (attempted merging values cake and tart)
181         Map<Integer, String> map =
182                 Stream.of( ...values: "cake", "biscuits", "tart")
183                         .collect(
184                                 Collectors.toMap(String::length,   // length is the key
185                                                  String::toString) // dessert name is the value
186                         );
187         System.out.println(map);
```

```
156         // To get around the duplicate keys issue, we can supply a merge function,
157         // whereby we append the colliding keys values together.
158         Map<Integer, String> map =
159                 Stream.of( ...values: "cake", "biscuits", "tart")
160                 .collect(
161                     Collectors.toMap(s -> s.length(),// length is the key
162                                      s -> s,         // dessert name is the value
163                                      (s1, s2) -> s1 + "," + s2)// Merge function - what to
164                                                                // do if we have duplicate keys
165                                                                //   - append the values
166                 );
167         System.out.println(map);// {4=cake,tart, 8=biscuits}
```

```
// The maps returned are not guaranteed. What if we wanted
// a TreeMap implementation so our keys would be sorted. The last argument
// caters for this.
TreeMap<String, Integer> map =
        Stream.of( ...values: "cake", "biscuits", "apple tart", "cake")
        .collect(
            Collectors.toMap(String::toString, // dessert name is the key
                             String::length,   // length is the value
                             (len1, len2) -> len1 +len2, // what to do if we have
                                                         // duplicate keys:
                                                         //    - add the *values*
                             TreeMap::new ));   // Supplier
System.out.println(map);// {apple tart=10, biscuits=8, cake=8} Note: cake maps to 8
System.out.println(map.getClass());// class java.util.TreeMap
```

```
Stream<String> names = Stream.of( ...values: "Martin", "Peter", "Tom", "Tom", "Ann");
Map<Integer, List<String>> map =
        names.collect(
            // passing in a Function that determines the
            // key in the Map
            Collectors.groupingBy(String::length) // name -> name.length()
        );
System.out.println(map);// {3=[Tom, Tom, Ann], 5=[Peter], 6=[Martin]}
```

```
Stream<String> names = Stream.of( ...values: "Martin", "Peter", "Tom", "Tom", "Ann");
Map<Integer, Set<String>> map =
        names.collect(
            Collectors.groupingBy(
                    String::length,     // key Function
                    Collectors.toSet()) // what to do with the values
        );
System.out.println(map);// {3=[Ann, Tom], 5=[Peter], 6=[Martin]}
System.out.println(map.getClass());// class java.util.HashMap
```

```
Stream<String> names = Stream.of( ...values: "Martin", "Peter", "Tom", "Tom", "Ann");
Map<Integer, List<String>> map =
        names.collect(
            Collectors.groupingBy(
                    String::length,
                    TreeMap::new,       // map type Supplier
                    Collectors.toList())// downstream collector
        );
System.out.println(map);// {3=[Tom, Tom, Ann], 5=[Peter], 6=[Martin]}
System.out.println(map.getClass());// class java.util.TreeMap
```

```
93      Stream<String> names = Stream.of( ...values: "Thomas", "Teresa",
94                                                   "Mike", "Alan", "Peter");
95      Map<Boolean, List<String>> map =
96              names.collect(
97                  // pass in a Predicate
98                  Collectors.partitioningBy(s -> s.startsWith("T"))
99              );
100     System.out.println(map);// {false=[Mike, Alan, Peter],
101                             // true=[Thomas, Teresa]}
```

```
59  Stream<String> names = Stream.of( ...values: "Alan", "Teresa", "Mike",
60                                               "Alan", "Peter"); // "Alan" here twice
61  Map<Boolean, Set<String>> map =
62          names.collect(
63              Collectors.partitioningBy(
64                  s -> s.length() > 4,// predicate
65                  Collectors.toSet()    )
66          );
67  System.out.println(map);// {false=[Mike, Alan], true=[Teresa, Peter]}
```

# Chapter 16: Streams: Advanced Concepts

```
141         // Stream<T> filter(Predicate)
142         // The filter() method returns a Stream with the elements that
143         // MATCH the given predicate.
144         Stream.of( ...values: "Canada", "Ireland", "Spain")
145                 .filter(country -> country.length() > 5)
146                 .forEach(System.out::print);// CanadaIreland
```

```
146     public static void doFilter() {
147
148         // Stream<T> filter(Predicate)
149         // The filter() method returns a Stream with the elements that
150         // MATCH the given predicate.
151         Stream.of( ...values: "Canada", "Ireland", "Spain")
152                 .filter(country -> country.length() > 5)
153                 .forEach(System.out::print);// CanadaIreland
154
155     }
```

```
131         // Stream<T> distinct()
132         // distinct() is a stateful intermediate operation
133         // Output: Before: eagle, After: eagle
134         //         Before: eagleBefore: EAGLE, After: EAGLE
135         Stream.of( ...values: "eagle", "eagle", "EAGLE")
136                 .peek(s -> System.out.print("Before: "+s))
137                 .distinct()
138                 .forEach(s -> System.out.print(", After: "+s + "\n"));
```

```java
public static void doDistinct() {

    // Stream<T> distinct()
    // distinct() is a stateful intermediate operation
    // Output: Before: eagle, After: eagle
    //         Before: eagleBefore: EAGLE, After: EAGLE
    Stream.of( ...values: "eagle", "eagle", "EAGLE")
            .peek(s -> System.out.print("Before: "+s))
            .distinct()
            .forEach(s -> System.out.print(", After: "+s + "\n"));

}
```

```java
    // Stream<T> limit(long maxSize)
    // limit is a short-circuiting stateful intermediate operation.
    // Lazy evaluation - 66, 77, 88 and 99 are not streamed as they
    // are not needed (limit of 2 i.e. 44 and 55).
    // Output:
    //  A - 11 A - 22 A - 33 A - 44 B - 44 C - 44 A - 55 B - 55 C - 55
    Stream.of( ...values: 11,22,33,44,55,66,77,88,99)
            .peek(n -> System.out.print(" A - "+n))
            .filter(n -> n > 40)
            .peek(n -> System.out.print(" B - "+n))
            .limit( maxSize: 2)
            .forEach(n -> System.out.print(" C - "+n));
```

```java
public static void doLimit() {

    // Stream<T> limit(long maxSize)
    // limit is a short-circuiting stateful intermediate operation.
    // Lazy evaluation - 66, 77, 88 and 99 are not streamed as they
    // are not needed (limit of 2 i.e. 44 and 55).
    // Output:
    //  A - 11 A - 22 A - 33 A - 44 B - 44 C - 44 A - 55 B - 55 C - 55
    Stream.of( ...values: 11,22,33,44,55,66,77,88,99)
            .peek(n -> System.out.print(" A - "+n))
            .filter(n -> n > 40)
            .peek(n -> System.out.print(" B - "+n))
            .limit( maxSize: 2)
            .forEach(n -> System.out.print(" C - "+n));
}
```

```
108         // <R> Stream<R> map(Function<T,R> mapper)
109         //     Function's functional method: R apply(T t);
110         Stream.of( ...values: "book", "pen", "ruler")
111                 .map(s -> s.length()) // String::length
112                 .forEach(System.out::print);// 435
```

```
111     public static void doMap() {
112
113         // <R> Stream<R> map(Function<T,R> mapper)
114         //     Function's functional method: R apply(T t);
115         Stream.of( ...values: "book", "pen", "ruler")
116                 .map(String::length) // s -> s.length()
117                 .forEach(System.out::print);// 435
118     }
```

```
98      List<String> nothing = List.of();
99      List<String> list1 = Arrays.asList("Sean");
100     List<String> list2 = Arrays.asList("Maike", "van", "Putten");
101     Stream<List<String>> streamOfLists = Stream.of(nothing, list1, list2);
102     streamOfLists.forEach(System.out::print); // [][Sean][Maike, van, Putten]
103     System.out.println(); // blank line to separate outputs
104
105     // flatMap(Function(T, R)) IN:T OUT:R
106     //  flatMap(Function(List<String>, Stream<String>))
107     streamOfLists = Stream.of(nothing, list1, list2);
108     streamOfLists.flatMap(list -> list.stream())
109             .forEach(System.out::print);// SeanMaikevanPutten
```

```java
public static void doFlatMap() {
    List<String> nothing = List.of();
    List<String> list1 = Arrays.asList("Sean");
    List<String> list2 = Arrays.asList("Maike", "van", "Putten");
    Stream<List<String>> streamOfLists = Stream.of(nothing, list1, list2);
    streamOfLists.forEach(System.out::print); // [][Sean][Maike, van, Putten]
    System.out.println(); // blank line to separate outputs

    // flatMap(Function(T, R)) IN:T OUT:R
    //  flatMap(Function(List<String>, Stream<String>))
    streamOfLists = Stream.of(nothing, list1, list2);
    streamOfLists.flatMap(list -> list.stream())
            .forEach(System.out::print);// SeanMaikevanPutten
}
```

```java
    // Stream<T> sorted(Comparator<T> comparator)
    // Output:
    //   Person{name=John, age=23}Person{name=Mary, age=25}
    Person john = new Person( name: "John",  age: 23);
    Person mary = new Person( name: "Mary",  age: 25);
    Stream.of(mary,john)
            .sorted(Comparator.comparing(p -> p.getAge())) // Person::getAge
            .forEach(System.out::print);
```

```java
public static void doSorted2() {

    // Stream<T> sorted(Comparator<T> comparator)
    // Output:
    //   Person{name=John, age=23}Person{name=Mary, age=25}
    Person john = new Person( name: "John",  age: 23);
    Person mary = new Person( name: "Mary",  age: 25);
    Stream.of(mary,john)
            .sorted(Comparator.comparing(Person::getAge)) // p -> p.getAge()
            .forEach(System.out::print);

}
```

```
44      // Stream<T> sorted()
45      // Stream<T> sorted(Comparator<T> comparator)
46      // Output:
47      //  0.Tim 1.Tim 0.Jim 1.Jim 0.Peter 0.Ann 1.Ann 0.Mary 2.Ann 3.Ann 2.Jim 3.Jim
48      Stream.of( ...values: "Tim", "Jim", "Peter", "Ann", "Mary")
49              .peek(name -> System.out.print(" 0."+name))    // Tim, Jim, Peter, Ann, Mary
50              .filter(name -> name.length() == 3)
51              .peek(name -> System.out.print(" 1."+name))    // Tim, Jim, Ann
52              .sorted()                                      // Tim, Jim, Ann (stored)
53              .peek(name -> System.out.print(" 2."+name))    // Ann, Jim
54              .limit( maxSize: 2)
55              .forEach(name -> System.out.print(" 3."+name));// Ann, Jim
```

```
42  public static void doSortedOther() {
43
44      // Stream<T> sorted()
45      // Stream<T> sorted(Comparator<T> comparator)
46      // Output:
47      //  0.Tim 1.Tim 0.Jim 1.Jim 0.Peter 0.Ann 1.Ann 0.Mary 2.Ann 3.Ann 2.Jim 3.Jim
48      Stream.of( ...values: "Tim", "Jim", "Peter", "Ann", "Mary")
49              .peek(name -> System.out.print(" 0."+name))    // Tim, Jim, Peter, Ann, Mary
50              .filter(name -> name.length() == 3)
51              .peek(name -> System.out.print(" 1."+name))    // Tim, Jim, Ann
52              .sorted()                                      // Tim, Jim, Ann (stored)
53              .peek(name -> System.out.print(" 2."+name))    // Ann, Jim
54              .limit( maxSize: 2)
55              .forEach(name -> System.out.print(" 3."+name));// Ann, Jim
56
57  }
```

```
25          int[] ia    = {1,2,3};
26          double[] da = {1.1, 2.2, 3.3};
27          long[] la   = {1L, 2L, 3L};
28
29          IntStream iStream1      = Arrays.stream(ia);
30          DoubleStream dStream1   = Arrays.stream(da);
31          LongStream lStream1     = Arrays.stream(la);
32          System.out.println(iStream1.count() + ", " +
33                  dStream1.count() + ", " + lStream1.count());// 3, 3, 3
34
35          IntStream iStream2       = IntStream.of( ...values: 1, 2, 3);
36          DoubleStream dStream2    = DoubleStream.of( ...values: 1.1, 2.2, 3.3);
37          LongStream lStream2      = LongStream.of( ...values: 1L, 2L, 3L);
38          System.out.println(iStream2.count() + ", " +
39                  dStream2.count() + ", " + lStream2.count());// 3, 3, 3
```

```
37      public static void creatingFinitePrimitiveStreams(){
38          int[] ia    = {1,2,3};
39          double[] da = {1.1, 2.2, 3.3};
40          long[] la   = {1L, 2L, 3L};
41
42          IntStream iStream1      = Arrays.stream(ia);
43          DoubleStream dStream1   = Arrays.stream(da);
44          LongStream lStream1     = Arrays.stream(la);
45          System.out.println(iStream1.count() + ", " +
46                  dStream1.count() + ", " + lStream1.count());// 3, 3, 3
47
48          IntStream iStream2       = IntStream.of( ...values: 1, 2, 3);
49          DoubleStream dStream2    = DoubleStream.of( ...values: 1.1, 2.2, 3.3);
50          LongStream lStream2      = LongStream.of( ...values: 1L, 2L, 3L);
51          System.out.println(iStream2.count() + ", " +
52                  dStream2.count() + ", " + lStream2.count());// 3, 3, 3
53      }
```

```
25          // DoubleStream generate(DoubleSupplier)
26          //     DoubleSupplier is a functional interface. Its
27          //     functional method is: double getAsDouble()
28          DoubleStream random    = DoubleStream.generate(() -> Math.random());
29          random.limit( maxSize: 5).forEach(System.out::println);
30
31          // IntStream iterate(int seed, IntUnaryOperator f)
32          //     IntUnaryOperator is a functional interface. Its
33          //     functional method is: int applyAsInt(int)
34          IntStream even = IntStream.iterate( seed: 2, (n) -> n + 2);
35          even.limit( maxSize: 5).forEach(System.out::println);
```

```java
public static void creatingInfinitePrimitiveStreams(){
    // DoubleStream generate(DoubleSupplier)
    //     DoubleSupplier is a functional interface. Its
    //     functional method is: double getAsDouble()
    DoubleStream random    = DoubleStream.generate(() -> Math.random());
    random.limit( maxSize: 5).forEach(System.out::println);

    // IntStream iterate(int seed, IntUnaryOperator f)
    //     IntUnaryOperator is a functional interface. Its
    //     functional method is: int applyAsInt(int)
    IntStream even = IntStream.iterate( seed: 2, (n) -> n + 2);
    even.limit( maxSize: 5).forEach(System.out::println);
}
```

```java
public static void maxMinAverage(){

    OptionalInt max = IntStream.of( ...values: 10, 20, 30)
            .max(); // terminal operation
    max.ifPresent(System.out::println);// 30

    OptionalDouble min = DoubleStream.of( ...values: 10.0, 20.0, 30.0)
            .min(); // terminal operation
    // NoSuchElementException is thrown if no value present
    System.out.println(min.orElseThrow());// 10.0

    OptionalDouble average = LongStream.of( ...values: 10L, 20L, 30L)
            .average(); // terminal operation
    System.out.println(average.orElseGet(() -> Math.random()));// 20.0
}
```

```java
    public static void usingSum(){

        IntStream is = IntStream.of( ...values: 4, 2, 3);
        System.out.println(is.sum());// 9

        // 1. Using Stream<T> and reduce(identity, accumulator)
        Stream<Integer> numbers = Stream.of( ...values: 1,2,3);
        System.out.println(numbers.reduce( identity: 0, (n1, n2) -> n1 + n2));// 6

        // 2. Using IntStream and sum()
        // IntStream mapToInt(ToIntFunction)
        //   ToIntFunction is a functional interface:
        //     int applyAsInt(T value);
        Stream<Integer> sInteger = Stream.of( ...values: 1,2,3);
        IntStream intS           = sInteger.mapToInt( n -> n);// unboxed
        System.out.println(intS.sum());// 6
```

```java
public static void stats(IntStream numbers){
    IntSummaryStatistics intStats =
            numbers.summaryStatistics();        // terminal operation
    System.out.println(intStats.getMin());      // 5 (2147483647 if nothing in stream)
    System.out.println(intStats.getMax());      // 20 (-2147483648 if nothing in stream)
    System.out.println(intStats.getAverage());  // 12.5 (0.0 if nothing in stream)
    System.out.println(intStats.getCount());    // 4 (0 if nothing in stream)
    System.out.println(intStats.getSum());      // 50 (0 if nothing in stream)
}
```

```java
public static void mappingObjectStreams(){
    // Stream<T> to Stream<T>
    Stream.of( ...values: "ash", "beech", "sycamore")
            // map(Function<T,R>)
            //    Function<T,R> => Function<String, String>
            //       String apply(String s)
            .map(tree -> tree.toUpperCase())
            .forEach(System.out::println);// ASH, BEECH, SYCAMORE

    // Stream<T> to DoubleStream
    DoubleStream dblStream = Stream.of( ...values: "ash", "beech", "sycamore")
            // mapToDouble(ToDoubleFunction<T>)
            //   ToDoubleFunction<T> is a functional interface:
            //      double applyAsDouble(T value) => double applyAsDouble(String tree)
            .mapToDouble(tree -> tree.length()); // upcast in background
    dblStream.forEach(System.out::println); // 3.0, 5.0, 8.0

    // Stream<T> to IntStream
    IntStream intStream    = Stream.of( ...values: "ash", "beech", "sycamore")
            // mapToInt(ToIntFunction<T>)
            //   ToIntFunction<T> is a functional interface:
            //      int applyAsInt(T value) => int applyAsInt(String tree)
            .mapToInt(tree -> tree.length());
    intStream.forEach(System.out::println); // 3, 5, 8

    // Stream<T> to LongStream
    LongStream longStream = Stream.of( ...values: "ash", "beech", "sycamore")
            // mapToLong(ToLongFunction<T>)
            //   ToLongFunction<T> is a functional interface:
            //      long applyAsLong(T value) => long applyAsLong(String tree)
            .mapToLong(tree -> tree.length()); // upcast in background
    longStream.forEach(System.out::println); // 3, 5, 8

}
```

```java
public static void mappingPrimitiveStreams(){

    // IntStream to Stream<T>
    Stream<String> streamStr = IntStream.of( ...values: 1, 2, 3)
            // mapToObj(IntFunction<R>)
            //    IntFunction is a functional interface:
            //       R apply(int value)
            .mapToObj(n -> "Number:"+ n);
    streamStr.forEach(System.out::println);// Number:1, Number:2, Number:3

    // IntStream to DoubleStream
    DoubleStream dblStream = IntStream.of( ...values: 1, 2, 3) // re-open closed stream
            // mapToDouble(IntToDoubleFunction)
            //   IntToDoubleFunction is a functional interface:
            //      double applyAsDouble(int value)
            .mapToDouble(n -> (double)n); // cast NOT necessary
    dblStream.forEach(System.out::println); // 1.0, 2.0, 3.0

    // IntStream to IntStream
    IntStream intStream = IntStream.of( ...values: 1, 2, 3)
            //  map(IntUnaryOperator)
            //    IntUnaryOperator is a functional interface:
            //       int applyAsInt(int)
            .map(n -> n*2);
    intStream.forEach(System.out::println);// 2, 4, 6

    // IntStream to LongStream
    LongStream longStream = IntStream.of( ...values: 1, 2, 3)
            // mapToLong(IntToLongFunction)
            //   IntToLongFunction is a functional interface:
            //      long applyAsLong(int value)
            .mapToLong(n -> (long)n); // cast NOT necessary
    longStream.forEach(System.out::println); // 1, 2, 3

}
```

```java
    public static void createOptionals(){
        Optional opt1 = Optional.empty();
//        System.out.println(opt1.get()); // NoSuchElementException
        opt1.ifPresent(o -> System.out.println("opt1: "+o)); // no exception

        Optional opt2 = Optional.of( value: 23);
//        Optional.of(null); // NullPointerException
        opt2.ifPresent(o -> System.out.println("opt2: "+o)); // opt2: 23

        Optional opt3 = Optional.ofNullable( value: 23);
        opt3.ifPresent(o -> System.out.println("opt3: "+o)); // opt3: 23

        Optional opt4 = Optional.ofNullable( value: null);
        opt4.ifPresent(o -> System.out.println("opt4: "+o));
        if(opt4.isEmpty()){
            System.out.println("opt4 is empty!");           // opt4 is empty!
        }
    }
```

```java
    public static void doOptionalAPI(){
       Optional<Double> valueInOptional = Optional.ofNullable( value: 60.0);
        if(valueInOptional.isPresent()){
            System.out.println(valueInOptional.get()); // 60.0
        }
        valueInOptional.ifPresent(System.out::println);// 60.0
        System.out.println(valueInOptional.orElse(Double.NaN)); // 60.0

        Optional<Double> emptyOptional = Optional.ofNullable( value: null);
        System.out.println(emptyOptional.orElse(Double.NaN)); // NaN
        System.out.println(emptyOptional.orElseGet(() -> Math.random())); // 0.8524556508038182
        System.out.println(emptyOptional.orElseThrow()); // NoSuchElementException
//        System.out.println(emptyOptional.orElseThrow(() ->
//                                new RuntimeException())); // RuntimeException
    }
```

```
35      public static void doOptionalPrimitiveAverage(){
36          OptionalDouble optAvg = IntStream.rangeClosed(1, 10).average();
37          optAvg.ifPresent(d -> System.out.println(d));// 5.5
38          System.out.println(optAvg.getAsDouble());// 5.5
39          double dblAvg = optAvg.orElseGet(() -> Double.NaN);
40          System.out.println(dblAvg);// 5.5
41
42          OptionalInt optInt = OptionalInt.of( value: 35);
43          int age = optInt.orElseGet(() -> 0);
44          System.out.println(age);// 35
45          System.out.println(optInt.getAsInt());// 35
46
47
48      }
```

```
100     public static void sequentialAddition() {
101         int sum = Stream.of( ...values: 10, 20, 30, 40, 50, 60) Stream<Integer>
102                         .mapToInt(Integer::intValue) IntStream
103                         .sum();
104         System.out.println("Sum == "+sum); // 210
105     }
106     public static void parallelAddition() {
107         int sum = Stream.of( ...values: 10, 20, 30, 40, 50, 60) Stream<Integer>
108                 .parallel() // Stream<T> method
109                 .mapToInt(Integer::intValue) IntStream
110                 .sum();
111         System.out.println("Sum == "+sum); // 210
112     }
```

```
73      public static int dbAction(int x){
74          try {
75              Thread.sleep( millis: 1000);
76          } catch (InterruptedException e) {
77              throw new RuntimeException(e);
78          }
79          return x;
80      }
81      public static void orderedSerialStreams(){
82          long atStart = System.currentTimeMillis();
83          List.of(10, 20, 30, 40, 50) List<Integer>
84                  .stream() Stream<Integer>
85                  .map(i -> dbAction(i))
86                  .forEach(i -> System.out.print(i + " "));
87          long howLong = (System.currentTimeMillis() - atStart) / 1000;
88          System.out.println("\nOperation took: "+howLong+" seconds.");
89      }
90      public static void unorderedParalleltreams(){
91          long atStart = System.currentTimeMillis();
92          List.of(10, 20, 30, 40, 50) List<Integer>
93                  .parallelStream() Stream<Integer>
94                  .map(i -> dbAction(i))
95                  .forEach(i -> System.out.print(i + " "));
96          long howLong = (System.currentTimeMillis() - atStart) / 1000;
97          System.out.println("\nOperation took: "+howLong+" seconds.");
98      }
```

```
90      public static void unorderedParalleltreams(){
91          long atStart = System.currentTimeMillis();
92          List.of(10, 20, 30, 40, 50) List<Integer>
93                  .parallelStream() Stream<Integer>
94                  .map(i -> dbAction(i))
95                  .forEachOrdered(i -> System.out.print(i + " "));
96      //              .forEach(i -> System.out.print(i + " "));
97          long howLong = (System.currentTimeMillis() - atStart) / 1000;
98          System.out.println("\nOperation took: "+howLong+" seconds.");
99      }
```

```java
public static void parallelReduction1(){
    int result = Stream.of( ...values: 1,2,3,4,5)
                .parallel()
                .reduce( identity: 0,
                    (n1, n2) -> { // accumulator
                        System.out.print(n1 + ", " + n2 + "\n");
                        return n1 - n2;
                    },
                    (subTask1, subTask2) -> { // combiner
                        System.out.print("\t" +subTask1 + ", " + subTask2 + "\n");
                        return subTask1 - subTask2;
                    });
    System.out.println(result); // 5
```

```java
            int result = Stream.of( ...values: 1,2,3,4,5)
                    .reduce( identity: 0,
                            (n1, n2) -> n1 - n2);
            System.out.println(result); // -15
```

```java
public static void concurrentCollection(){
    var names = Stream.of( ...values: "John","Mary","Mike", "Paula")
            .parallel();
    ConcurrentMap<Character, String> map =
            names.collect(Collectors.toConcurrentMap(
                    name -> name.charAt(0),  // key
                    name -> name,            // value
                    (name1, name2) -> name1 + ", "+ name2));// key collisions
    System.out.println(map); // {P=Paula, J=John, M=Mike, Mary}
    System.out.println(map.getClass()); // class java.util.concurrent.ConcurrentHashMap
}
```

# Chapter 17: Concurrency

