# Control Flow

## Overview of Control Flow Statements

- Selection statements: **if**, **if-else**, and **switch**.
- Iteration statements: **while**, **do-while**, ***basic* for**, and ***enhanced* for**.
- Transfer statements: **break**, **continue**, **return**, **try-catch-finally** , **throw**, and **assert**.

## Selection Statements

- simple **if** statement
- **if-else** statement
- **switch** statement

# The Simple if Statement

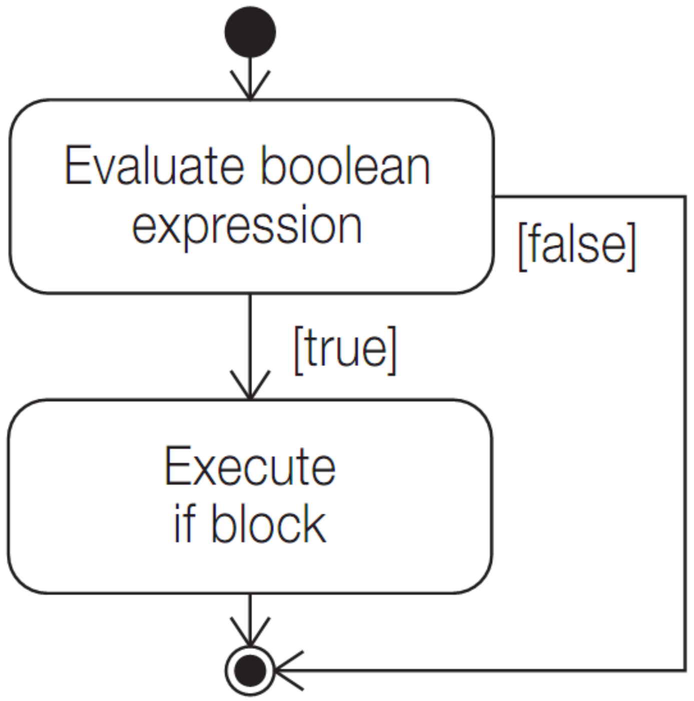
- The simple if statement has the following syntax:

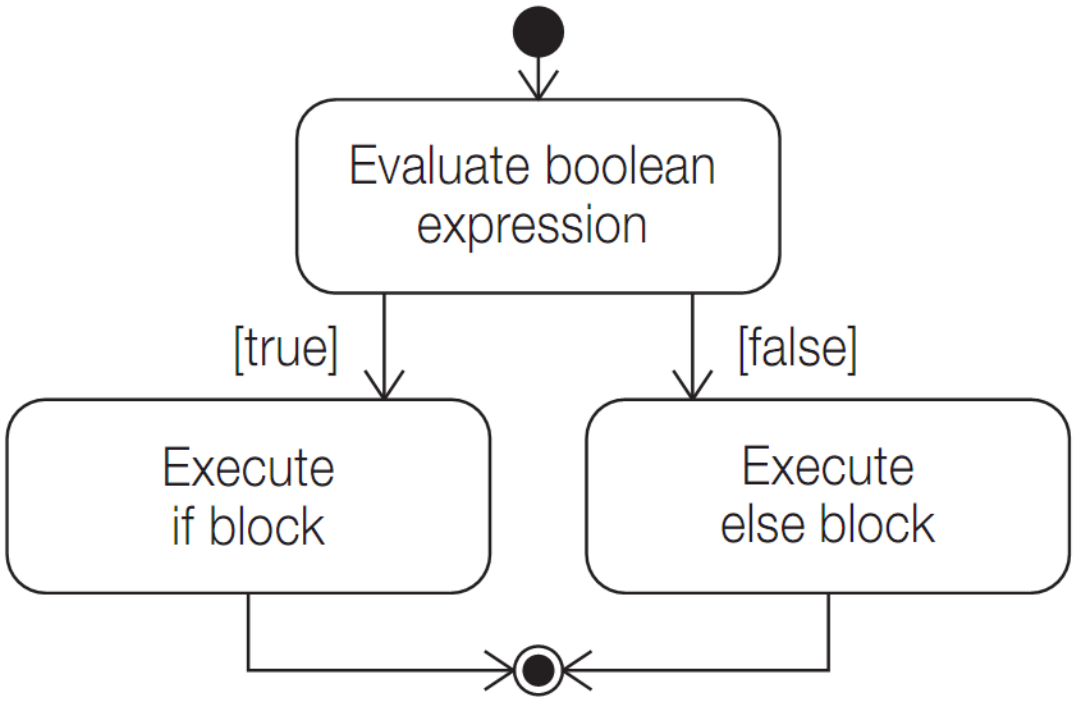
**`if`** (<conditional expression>) <statement>

Examples:

```
if (emergency) // emergency is a boolean variable
     operate();
if (temperature > critical)
     soundAlarm();
```

## Activity Diagram for if Statements





## The Simple if Statement

Note that <statement> can be a block, and the block notation is necessary if more than one statement is to be executed when the <conditional expression> is true.

```
if (catIsAway()) {      // Block
  getFishingRod();
  goFishing();
}
```

## The Simple if Statement

Note that the if block can be any valid statement. In particular, it can be the empty statement (;) or the empty block ({}). A common programming error is an inadvertent use of the empty statement.

```
if (emergency); // Empty if block
  operate();
// Executed regardless of whether it was an
// emergency or not.
```

# The if-else Statement

The if-else statement is used to decide between two actions, based on a condition. It has the following syntax:

```
if (<conditional expression>)
    <statement1>
else
    <statement2>
```

## The if-else Statement examples

```
if (emergency)
  operate();
else
  joinQueue();
if (temperature > critical)
  soundAlarm();
else
  businessAsUsual();
```

```
if (catIsAway()) {
  getFishingRod();
  goFishing();
} else
  playWithCat();
```

# if statements can be nested

```
if (temperature >= upperLimit) {          //(1)
  if (danger)                   // (2) Simple if.
    soundAlarm();
  if (critical)                 // (3)
    evacuate();
  else // Goes with if at (3).
    turnHeaterOff();
} else // Goes with if at (1).
    turnHeaterOn();
```

# Use of block notation {}

The use of the block notation, {}, can be critical to the execution of if statements.

```
// (A):
if (temperature > upperLimit) {           // (1) Block notation.
  if (danger) soundAlarm();               // (2)
} else                                    // Goes with if at (1).
  turnHeaterOn();
// (B):
if (temperature > upperLimit)             // (1) Without block notation.
  if (danger) soundAlarm();               // (2)
else turnHeaterOn();                      // Goes with if at (2).
// (C):
if (temperature > upperLimit)             // (1)
  if (danger)                             // (2)
    soundAlarm();
  else                                    // Goes with if at (2).
    turnHeaterOn();
```

# Cascading if-else statements

```
if (temperature >= upperLimit) {                  // (1)
  soundAlarm();
  turnHeaterOff();
} else if (temperature < lowerLimit) {            // (2)
  soundAlarm();
  turnHeaterOn();
} else if (temperature == (upperLimit+lowerLimit)/2) {// (3)
      doingFine();
} else                                            // (4)
  noCauseToWorry();
```

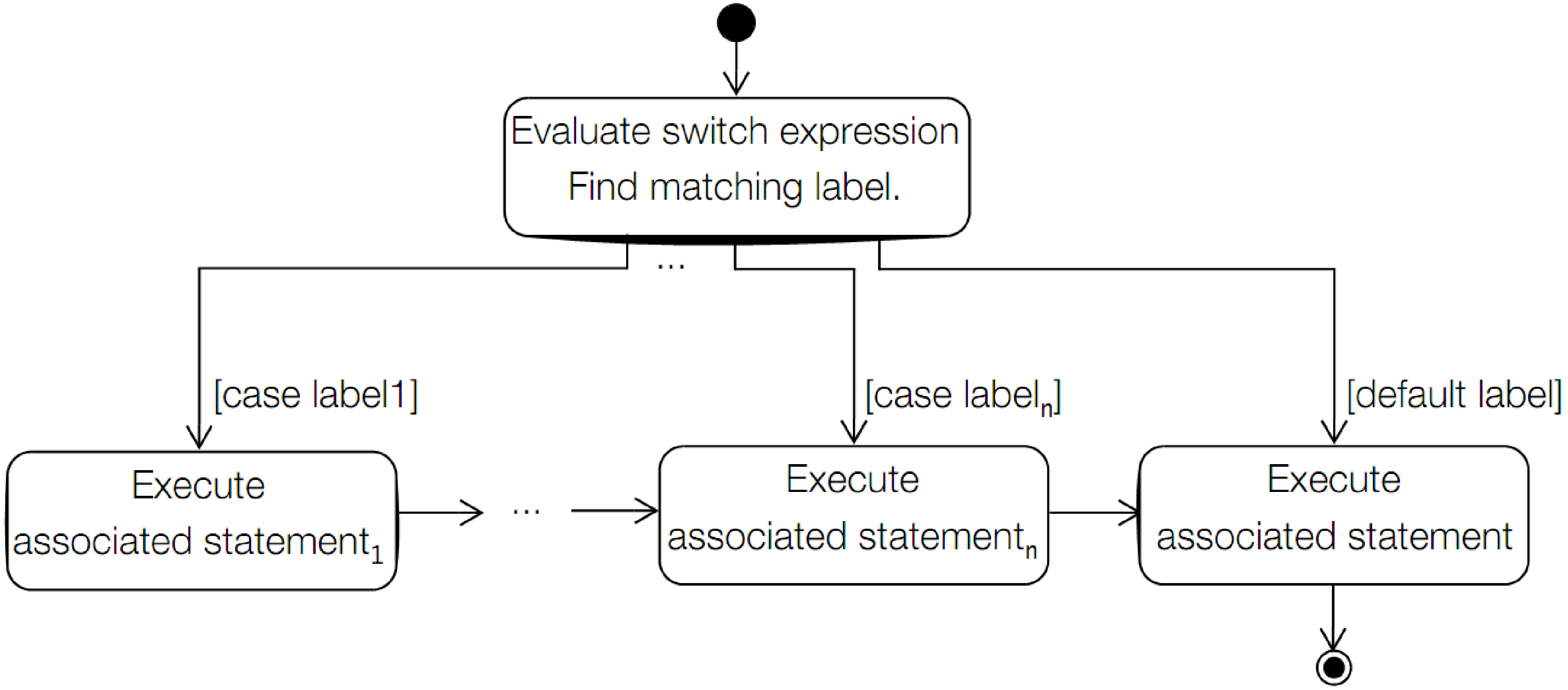
## The switch Statement

```
switch (<switch expression>) {
   case label1: <statement1>
   case label2: <statement2>
   ...
   case labeln: <statementn>
   default: <statement>
} // end switch
```

# Execution of the switch statement

- The  switch expression is evaluated first. If the value is a wrapper type, an unboxing conversion is performed.
- The value of the switch expression is compared with the case labels. Control is transferred to the <statement$_i$> associated with the case label that is equal to the value of the switch expression. After execution of the associated statement, control *falls through* to the *next* statement unless appropriate action is taken.
- If no case label is equal to the value of the switch expression, the statement associated with the default label is executed.

# Activity Diagram for a switch Statement


Evaluate switch expression
Find matching label.
...
[case label1]
[case label$_n$]
[default label]
Execute associated statement$_1$
...
Execute associated statement$_n$
Execute associated statement

## Fall Through in a switch Statement

```
public class Advice {
  public final static int LITTLE_ADVICE  = 0;
  public final static int MORE_ADVICE    = 1;
  public final static int LOTS_OF_ADVICE = 2;
  public static void main(String[] args) {
    dispenseAdvice(LOTS_OF_ADVICE);
  }
  public static void dispenseAdvice(int howMuchAdvice) {
    switch(howMuchAdvice) {                         // (1)
      case LOTS_OF_ADVICE:
        System.out.println("See no evil.");  // (2)
      case MORE_ADVICE:
        System.out.println("Speak no evil.");// (3)
      case LITTLE_ADVICE:
        System.out.println("Hear no evil."); // (4)
        break;                                      // (5)
      default:
        System.out.println("No advice.");    // (6)
    }
  }
}
```

## Using break in a switch Statement

```
public static String digitToString(char digit) {
    String str = "";
    switch(digit) {
      case ’1’: str = "one";   break;
      case ’2’: str = "two";   break;
      case ’3’: str = "three"; break;
      case ’4’: str = "four";  break;
      case ’5’: str = "five";  break;
      case ’6’: str = "six";   break;
      case ’7’: str = "seven"; break;
      case ’8’: str = "eight"; break;
      case ’9’: str = "nine";  break;
      case ’0’: str = "zero";  break;
      default:  System.out.println(digit + " is not a digit!");
    }
    return str;
  }
```

# Nested switch Statement

```
public class Seasons {
  public static void main(String[] args) {
    int monthNumber = 11;
    switch(monthNumber) {              // (1) Outer
      case 12: case 1: case 2:
        System.out.println("Snow in the winter.");
        break;
      case 3: case 4: case 5:
        System.out.println("Green grass in the spring.");
        break;
      case 6: case 7: case 8:
        System.out.println("Sunshine in the summer.");
        break;
      case 9: case 10: case 11:     // (2)
        switch(monthNumber) { // Nested switch    (3) Inner
```

# Nested switch Statement

```
        switch(monthNumber) { // Nested switch    (3) Inner
           case 10:
              System.out.println("Halloween.");
              break;
           case 11:
              System.out.println("Thanksgiving.");
              break;
         } // end nested switch
         // Always printed for case labels 9, 10, 11
         System.out.println("Yellow leaves in the fall.");     // (4)
         break;
       default:
         System.out.println(monthNumber + " is not a valid month.");
     }
   }
}
```

# Iteration Statements

Java provides four language constructs for loop construction:

- the **while** statement
- the **do-while** statement
- the ***basic*** **for** statement
- the ***enhanced*** **for** statement

## The while Statement

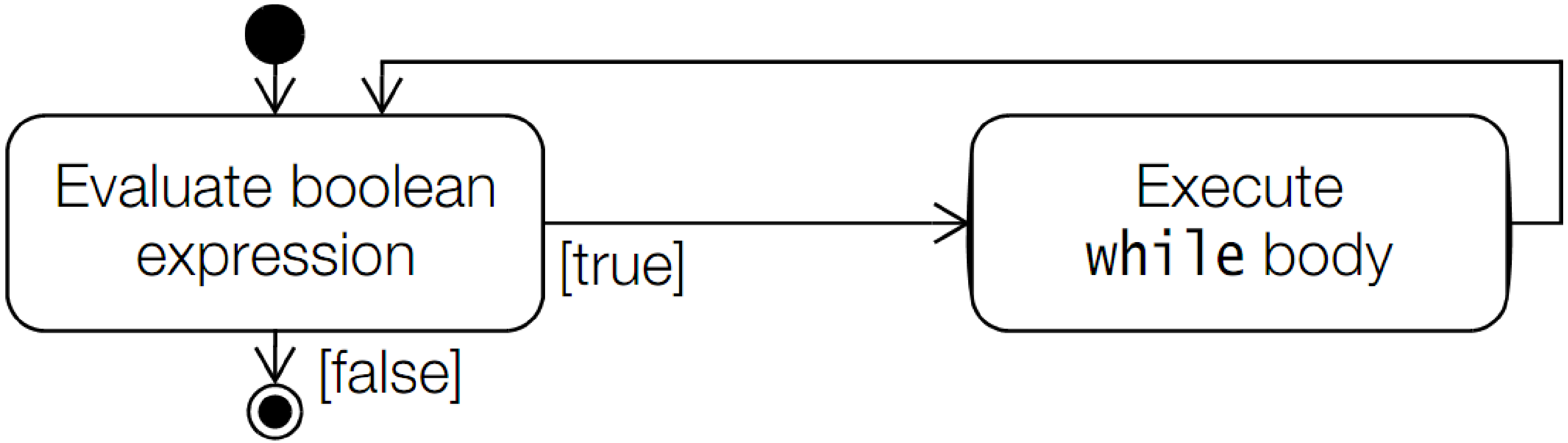
The syntax of the while loop is

```
while (<loop condition>)
    <loop body>
```

The <loop condition> is evaluated before executing the <loop body>. The while statement executes the <loop body> as long as the <loop condition> is true.

When the <loop condition> becomes false, the loop is terminated and execution continues with the statement immediately following the loop.

## Activity Diagram for the **`while`** Statement



The while statement is normally used when the number of iterations is not known.

```
while (noSignOfLife())
   keepLooking();
```

# The while Statement (warning)

Since the <loop body> can be any valid statement, inadvertently terminating each line with the empty statement (;) can give unintended results.

Always using a block statement, { ... }, as the <loop body> helps to avoid such problems.

```
while (noSignOfLife());//Empty statement as loop body!
  keepLooking();         // Statement not in the loop body.
```

# The do-while Statement

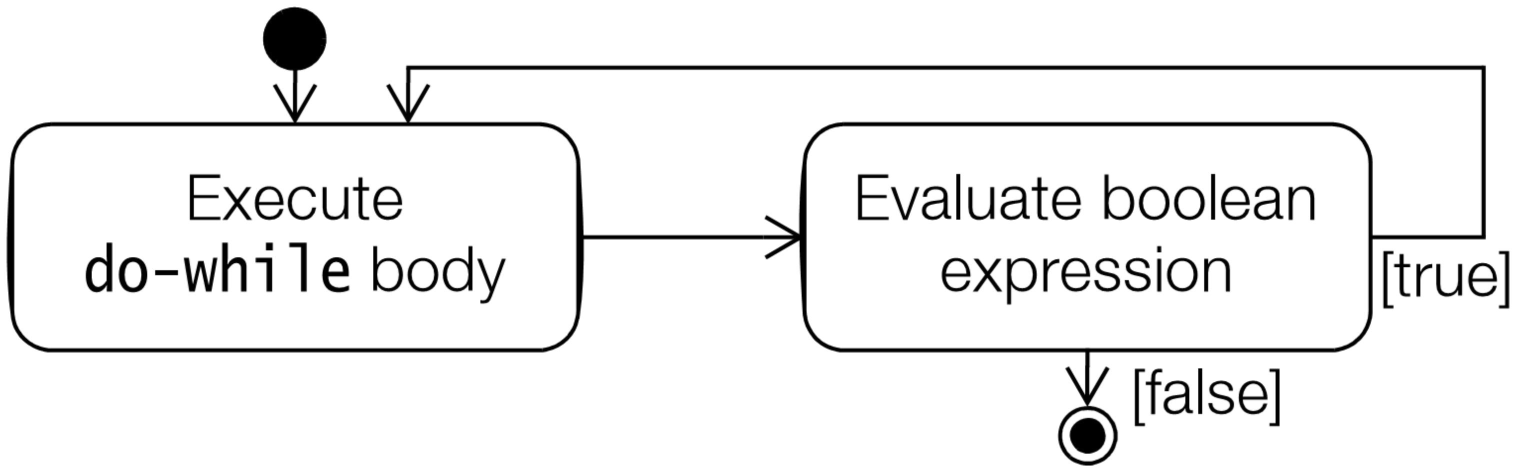
The syntax of the do-while loop is

```
do
<loop body>
while (<loop condition>);
```

The <loop condition> is evaluated after executing the <loop body>. The value of the <loop condition> is subjected to unboxing if it is of the type Boolean. The do-while statement executes the <loop body> until the <loop condition> becomes false.

When the <loop condition> becomes false, the loop is terminated and execution continues with the statement immediately following the loop.

## Activity Diagram for the do-while Statement



# while and do-while

The <loop body> in a do-while loop is invariably a statement block. It is instructive to compare the while and the do-while loops.

In the examples below, the mice might never get to play if the cat is not away, as in the loop at (1). The mice do get to play at least once (at the peril of losing their life) in the loop at (2).

```
while (cat.isAway()) {        // (1)
  mice.play();
}
do {                          // (2)
  mice.play();
} while (cat.isAway());
```

## The **`for(;;)`** Statement

The for(;;) loop is the most general of all the loops. It is mostly used for counter-controlled loops, i.e., when the number of iterations is known beforehand.

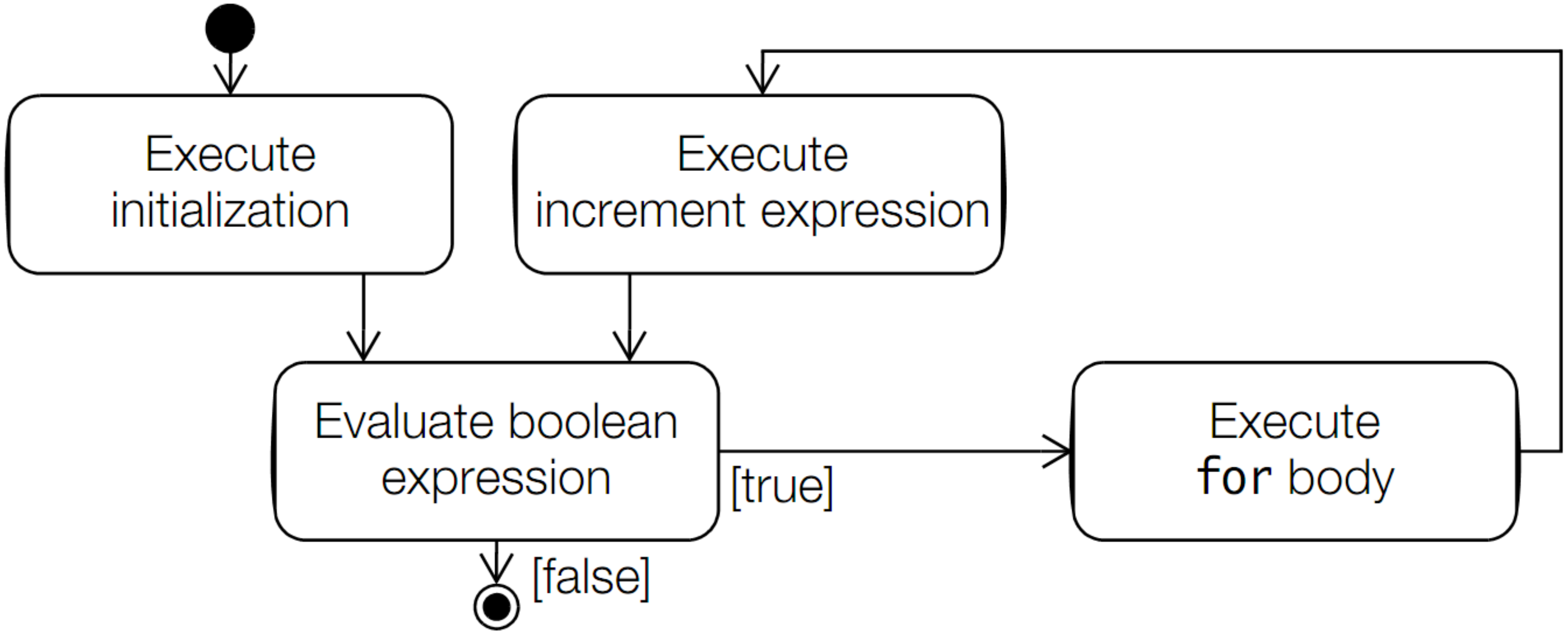
The syntax of the loop is as follows:

```
for (<initialization>; <loop condition>;
                                  <increment expression>)
   <loop body>
```

# The semantics of the **`for(;;)`** loop

```
<initialization>
while (<loop condition>) {
<loop body>
<increment expression>
}
```

# Activity Diagram for the for Statement



# for statement examples

```
int sum = 0;
int[] array = {12, 23, 5, 7, 19};
for (int index = 0; index < array.length; index++) // (1)
  sum += array[index];

for (int i = 0, j = 1, k = 2; ... ; ...) ...;       // (2)

for (int i = 0, String str = "@"; ... ; ...) ...;  // (3)
//Compile time error.

int i, j, k; // Variable declaration
for (i = 0, j = 1, k = 2; ... ; ...) ...;          // (4)
//Only initialization
```

# for statement examples

Declaration statements cannot be mixed with expression statements in the <initialization> section, as is the case at (5) in the following example. Factoring out the variable declaration, as at (6), leaves a legal comma-separated list of expression statements only.

```
// (5) Not legal and ugly:
for (int i = 0, System.out.println("not legal!"); flag; i++) { //Error!
  // loop body
}

// (6) Legal, but still ugly:
int i;       // declaration factored out.
for (i = 0, System.out.println("legal!"); flag; i++) {// OK.
  // loop body
}
```

## for statement examples

The <increment expression> can also be a comma-separated list of expression statements. The following code specifies a for(;;) loop that has a comma-separated list of three variables in the <initialization> section, and a comma-separated list of two expressions in the <increment expression> section:

# for statement examples

```
// Legal usage but not recommended.
int[][] sqMatrix = { {3, 4, 6}, {5, 7, 4}, {5, 8, 9} };
for (int i = 0,
     j = sqMatrix[0].length - 1,
     asymDiagonal = 0;                    // initialization
     i < sqMatrix.length;                 // loop condition
     i++, j--) // increment expression
  asymDiagonal += sqMatrix[i][j];        // loop body
```

# `for(;;)` statement

All sections in the **`for`**`(;;)` header are optional. Any or all of them can be left empty, but the two semicolons are mandatory. In particular, leaving out the <loop condition> signifies that the loop condition is true.

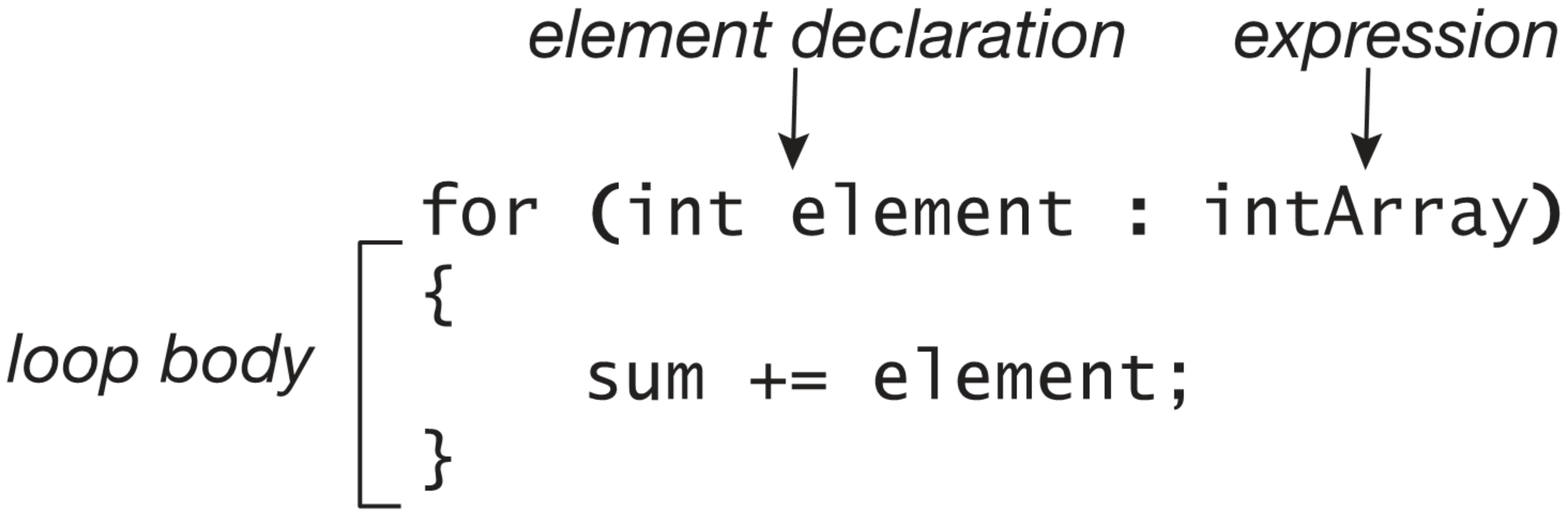
The “**crab**”, **`(;;)`**, is commonly used to construct an infinite loop, where termination is presumably achieved through code in the loop body (see next section on transfer statements):

```
for (;;) Java.programming();          // Infinite loop
```

## The for(:) Statement

The enhanced for loop is convenient when we need to iterate over an array or a collection, especially when some operation needs to be performed on each element of the array or collection.

# The for(:) Statement



```
for (int element : intArray)
{
      sum += element;
}
```

The element declaration specifies a local variable that can be assigned a value of the element type of the array. This assignment might require either a boxing or an unboxing conversion.

## The for(:) Statement

The element variable is local to the loop block and is not accessible after the loop terminates.

Also, changing the value of the current variable does not change any value in the array.

The loop body, which can be a simple statement or a statement block, is executed for each element in the array and there is no danger of any out-of-bounds errors.

# Transfer Statements

Java provides six language constructs for transferring control in a program:

- break
- continue
- return
- try-catch-finally
- throw
- assert

# Labeled Statements

A statement may have a label.

<label> : <statement>

A label is any valid identifier and it always immediately precedes the statement.

Label names exist in their own name space, so that they do not conflict with names of packages, classes, interfaces, methods, fields, and local variables.

# Labeled Statements

A statement can have multiple labels:

```
LabelA: LabelB:
System.out.println("Mutliple labels. Use judiciously.");
```

A declaration statement cannot have a label:

```
L0: int i = 0; // Compile time error.
```

A labeled statement is executed as if it was unlabeled, unless it is the break or continue statement.

## The break Statement

The break statement comes in two forms: the unlabeled and the labeled form.

```
break;          // the unlabeled form
break <label>; // the labeled form
```

## Unlabeled **break**

The unlabeled break statement terminates loops (for(;;), for(:), while, do-while)

and switch statements, and transfers control out of the current context (i.e., the closest enclosing block).

The rest of the statement body is skipped, and execution continues after the enclosing statement.

## Labeled **break**

A labeled break statement can be used to terminate any labeled statement that contains the break statement. Control is then transferred to the statement following the enclosing labeled statement. In the case of a labeled block, the rest of the block is skipped and execution continues with the statement following the block:

## Labeled **break**

```
out:
{ // (1) Labeled block
  // ...
  if (j == 10) break out;
  // (2) Terminate block. Control to (3).
  System.out.println(j);
  // Rest of the block not executed if j == 10.
  // ...
}
// (3) Continue here.
```

# The **`continue`** Statement

Like the break statement, the continue statement also comes in two forms: the unlabeled and the labeled form.

```
continue;                // the unlabeled form
continue <label>;        // the labeled form
```

# The **`continue`** Statement

The continue statement can only be used in a **`for`**`(;;)`, **`for`**`(:)`, **`while`**, or **`do-while`** loop to prematurely stop the current iteration of the loop body and proceed with the next iteration, if possible.

# The `continue` Statement

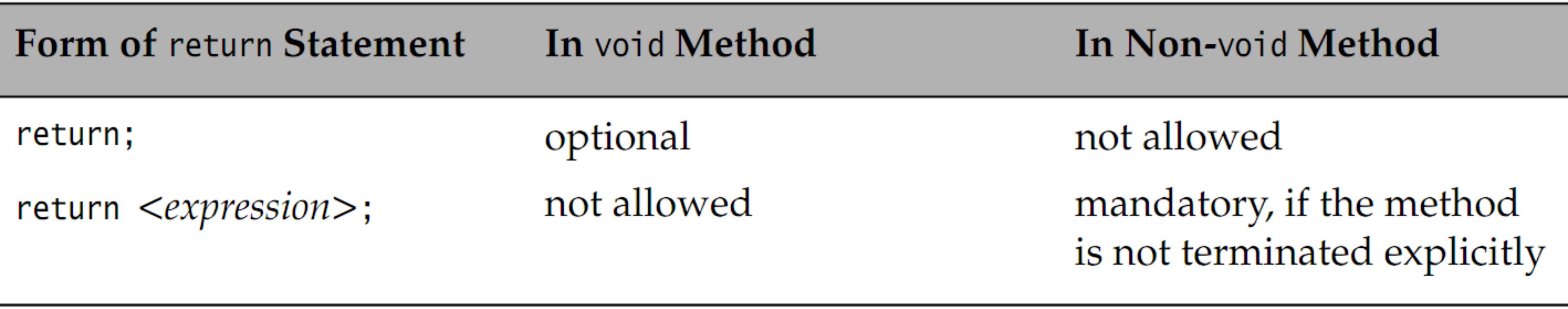
- In the case of the while and do-while loops, the rest of the loop body is skipped, that is, stopping the current iteration, with execution continuing with the <loop condition>.
- In the case of the for(;;) loop, the rest of the loop body is skipped, with execution continuing with the <increment expression>.

# The **`return`** Statement

The return statement is used to stop execution of a method and transfer control back to the calling code (also called the caller).

The usage of the two forms of the return statement is dictated by whether it is used in a void or a non-void method

# The **`return`** Statement

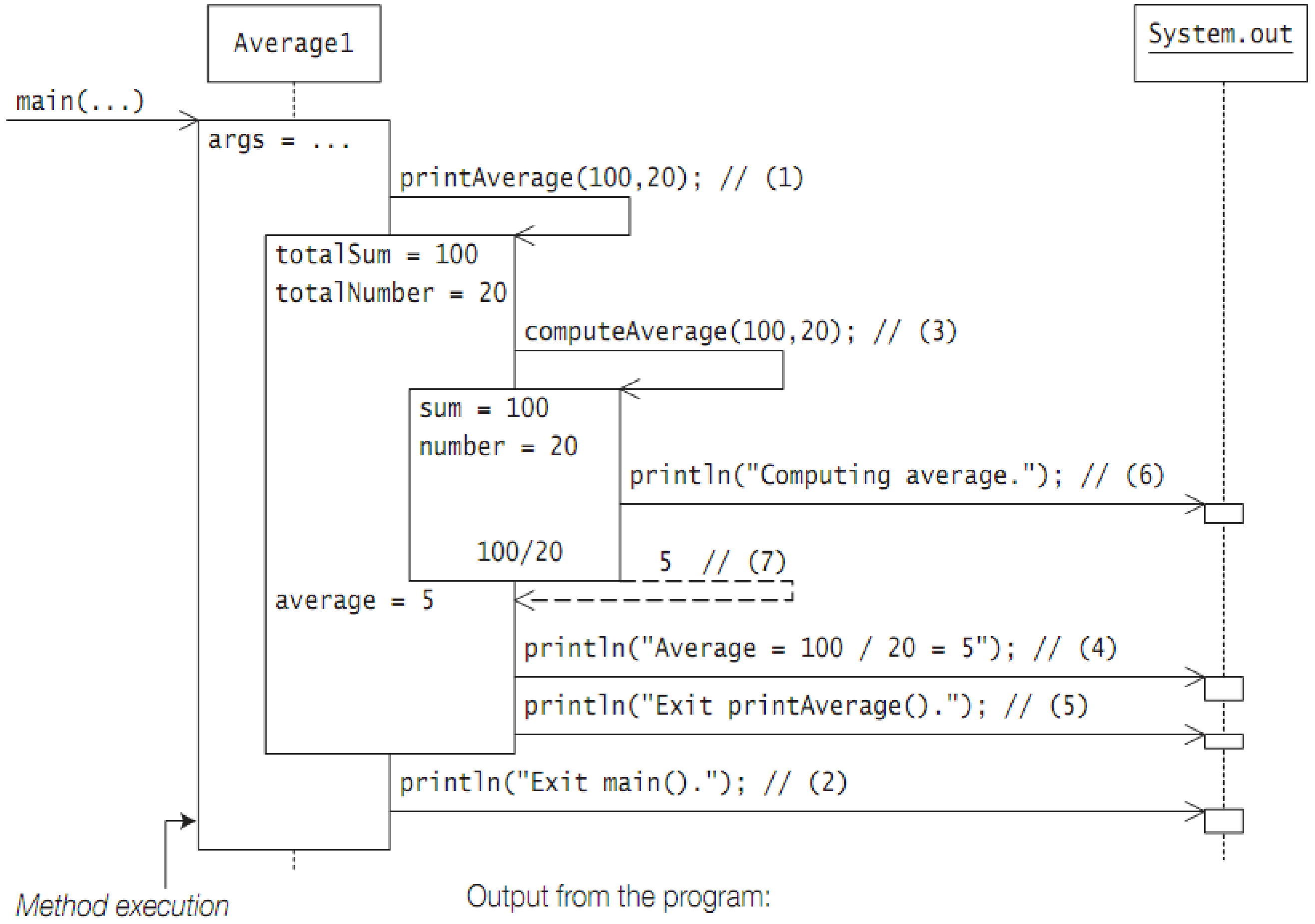


| Form of `return` Statement | In `void` Method | In Non-`void` Method |
|---|---|---|
| `return;` | optional | not allowed |
| `return` <*expression*>`;` | not allowed | mandatory, if the method is not terminated explicitly |

# Stack-Based Execution and Exception Propagation

```
public class Average1 {
  public static void main(String[] args) {
    printAverage(100, 20);                                    // (1a)
    System.out.println("Exit main().");                       // (2)
  }
  public static void printAverage(int totalSum, int totalNumber) {
    int average = computeAverage(totalSum, totalNumber);      // (3)
    System.out.println("Average = " +                         // (4)
        totalSum + " / " + totalNumber + " = " + average);
    System.out.println("Exit printAverage().");               // (5)
  }
  public static int computeAverage(int sum, int number) {
    System.out.println("Computing average.");                 // (6)
    return sum/number;                                        // (7)
  }
}
```

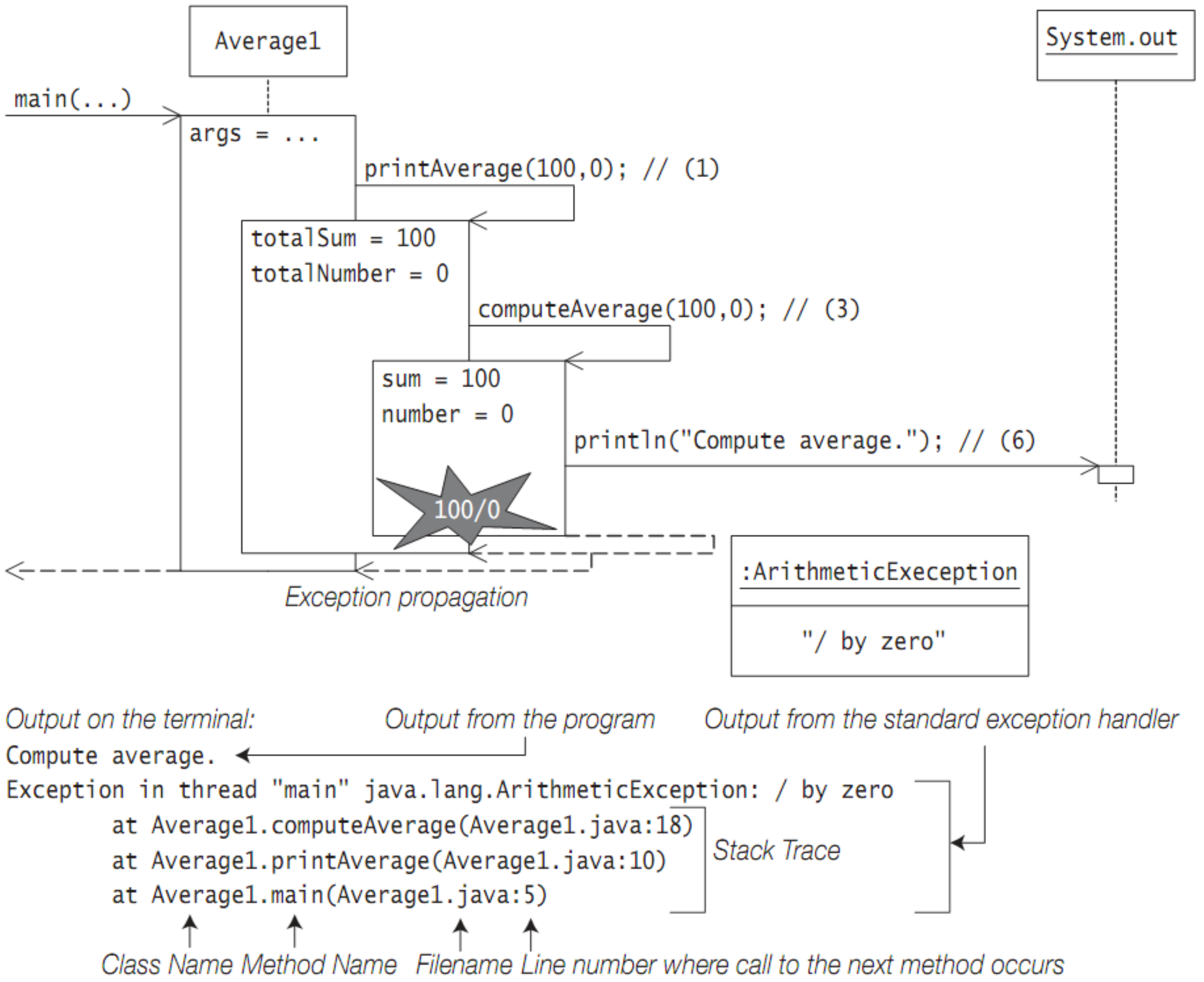
## Normal execution:


Average1
System.out
main(...)
args = ...
printAverage(100,20); // (1)
totalSum = 100
totalNumber = 20
computeAverage(100,20); // (3)
sum = 100
number = 20
println("Computing average."); // (6)
100/20
5 // (7)
average = 5
println("Average = 100 / 20 = 5"); // (4)
println("Exit printAverage()."); // (5)
println("Exit main()."); // (2)
Method execution

Output from the program:

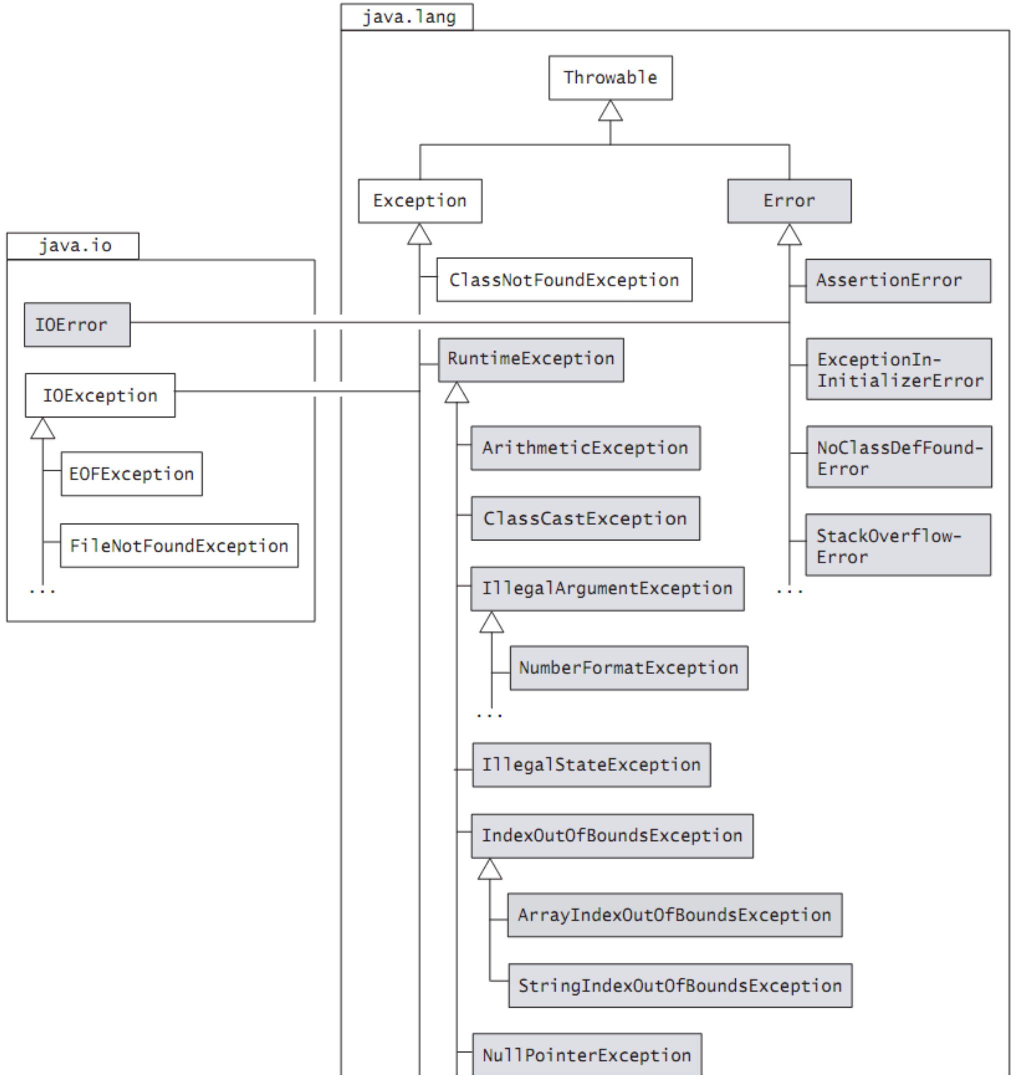
```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

## Exception propagation:



```
Compute average.
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Average1.computeAverage(Average1.java:18)
        at Average1.printAverage(Average1.java:10)
        at Average1.main(Average1.java:5)
```

# Exception Types



# Exceptions

In dealing with throwables, it is important to recognize situations under which particular throwables can occur, and the source that is responsible for throwing them.

By source here we mean:

- either it is the JVM that is responsible for throwing the throwable, or
- that the throwable is explicitly thrown programmatically by the code in the application or any API used by the application.

## The **Exception** Class

The class Exception represents exceptions that a program would normally want to catch.

Its subclass RuntimeException represents many common programming errors that can manifest at runtime.

Other subclasses of the Exception class define other categories of exceptions, e.g., I/O-related exceptions in the java.io package (IOException, FileNotFoundException, EOFException, IOError).

## ClassNotFoundException

The subclass ClassNotFoundException signals that the JVM tried to load a class by its string name, but the class could not be found. A typical example of this situation

is when the class name is misspelled while starting program execution with the java command.

The source in this case is the JVM throwing the exception to signal that the class cannot be found and therefore execution cannot be started.

# The RuntimeException Class

`ArithmeticException`

`ArrayIndexOutOfBoundsException`

`ClassCastException`

`IllegalArgumentException` and `NumberFormatException`

`IllegalStateException`

`NullPointerException`

## The Error Class

AssertionError

ExceptionInInitializerErro

IOError

NoClassDefFoundError

StackOverflowError

# Checked and Unchecked Exceptions

Except for **RuntimeException**, **Error**, and their subclasses, all exceptions are called checked exceptions.

The compiler ensures that if a method can throw a checked exception, directly or indirectly, the method must explicitly deal with it.

The method must either catch the exception and take the appropriate action, or pass the exception on to its caller

## Unchecked Exceptions

Exceptions defined by Error and RuntimeException classes and their subclasses are known as unchecked exceptions, meaning that a method is not obliged to deal with these kinds of exceptions.

- irrecoverable (exemplified by the Error)
- programming errors (exemplified by the RuntimeException)

## Defining New Exceptions

New exceptions are usually defined to provide fine-grained categorization of error situations, instead of using existing exception classes with descriptive detail messages to differentiate between the situations.

# Defining New Exceptions

New exceptions can either extend the Exception class directly or one of its checked subclasses, thereby making the new exceptions checked, or the RuntimeException class to create new unchecked exceptions.

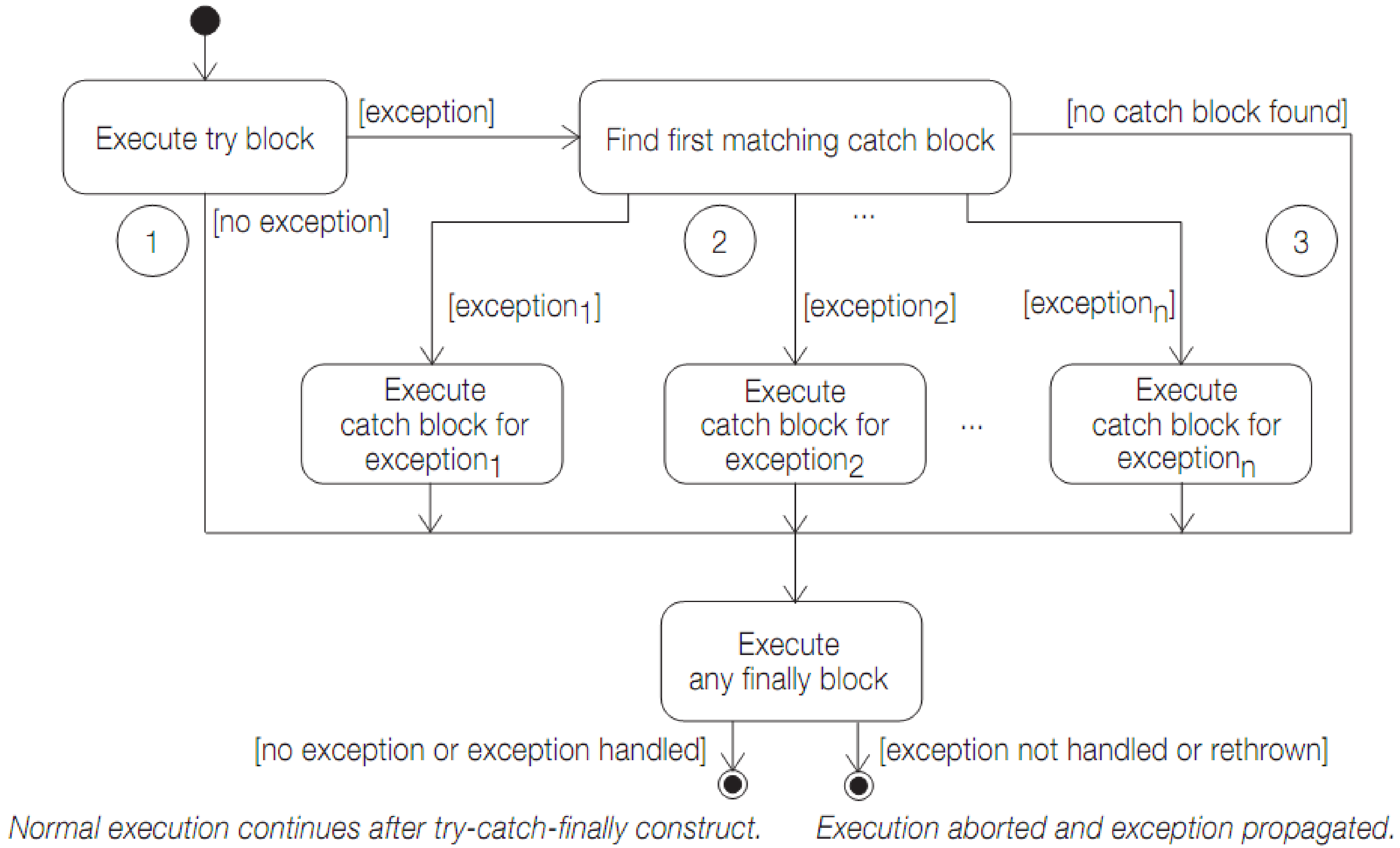
As exceptions are defined by classes, they can declare fields and methods, thus providing more information as to their cause and remedy when they are thrown and caught.

# Exception Handling: try, catch, and finally

```
try {                                        // try block
  <statements>
} catch (<exception type_1> <parameter_1>) { // catch block
  <statements>
}
...
  catch (<exception type_n> <parameter_n>) { // catch block
  <statements>
} finally {     // finally block
  <statements>
}
```

# try, catch, and finally

- For each try block there can be zero or more catch blocks, but only one finally block.
- The catch blocks and the finally block must always appear in conjunction with a try block, and in the right order.
- A try block must be followed by at least one catch block or a finally block must be specified.


Execute try block
[exception]
Find first matching catch block
[no catch block found]
1
[no exception]
2
...
3
[exception1]
[exception2]
[exceptionn]
Execute catch block for exception1
Execute catch block for exception2
...
Execute catch block for exceptionn
Execute any finally block
[no exception or exception handled]
[exception not handled or rethrown]
Normal execution continues after try-catch-finally construct.
Execution aborted and exception propagated.

## The try Block

The try block establishes a context for exception handling. Termination of a try block occurs as a result of encountering an exception, or from successful execution of the code in the try block.

The catch blocks are skipped for all normal exits from the try block where no exceptions were raised, and control is transferred to the finally block if one is specified.

For all exits from the try block resulting from exceptions, control is transferred to the catch blocks—if any such blocks are specified—to find a matching catch block.

If no catch block matches the thrown exception, control is transferred to the finally block if one is specified.

## The catch Block

A catch block can only catch the thrown exception if the exception is assignable to the parameter in the catch block.

The code of the first such catch block is executed and all other catch blocks are ignored.

On exit from a catch block, normal execution continues unless there is any pending exception that has been thrown and not handled.

If this is the case, the method is aborted and the exception is propagated up the runtime stack as explained earlier.

## The catch Block

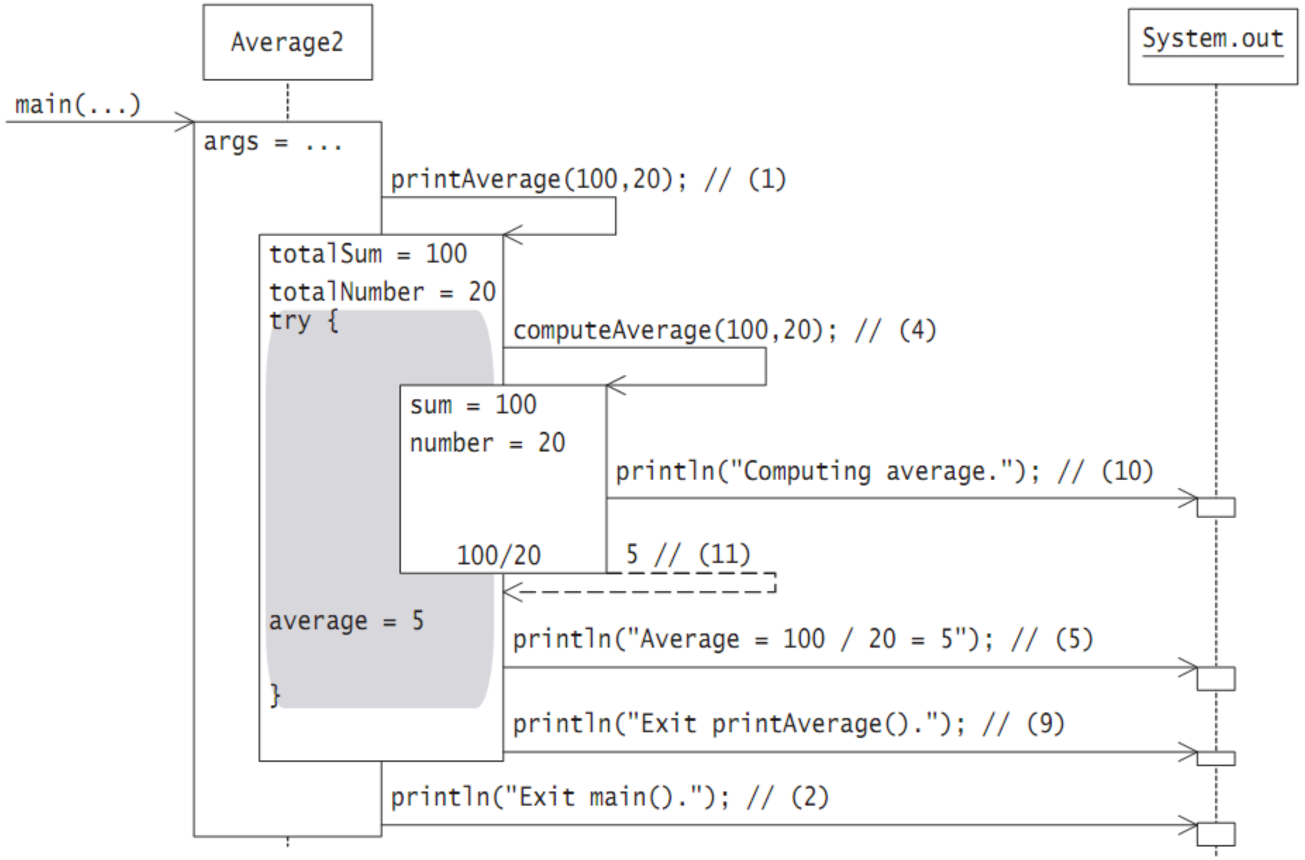
After a catch block has been executed, control is always transferred to the finally block if one is specified.

This is always true as long as there is a finally block, regardless of whether the catch block itself throws an exception.

# Exception handling example

```
public class Average2 {
  public static void main(String[] args) {
    printAverage(100, 20);                                              // (1)
    System.out.println("Exit main().");                                 // (2)
  }
  public static void printAverage(int totalSum, int totalNumber) {
    try {                                                               // (3)
      int average = computeAverage(totalSum, totalNumber);              // (4)
      System.out.println("Average = " +                                 // (5)
          totalSum + " / " + totalNumber + " = " + average);
    } catch (ArithmeticException ae) {                                   // (6)
      ae.printStackTrace();                                              // (7)
      System.out.println("Exception handled in printAverage().");        // (8)
    }
    System.out.println("Exit printAverage().");                          // (9)
  }
  public static int computeAverage(int sum, int number) {
    System.out.println("Computing average.");                            // (10)
    return sum/number;                                                   // (11)
  }
}
```

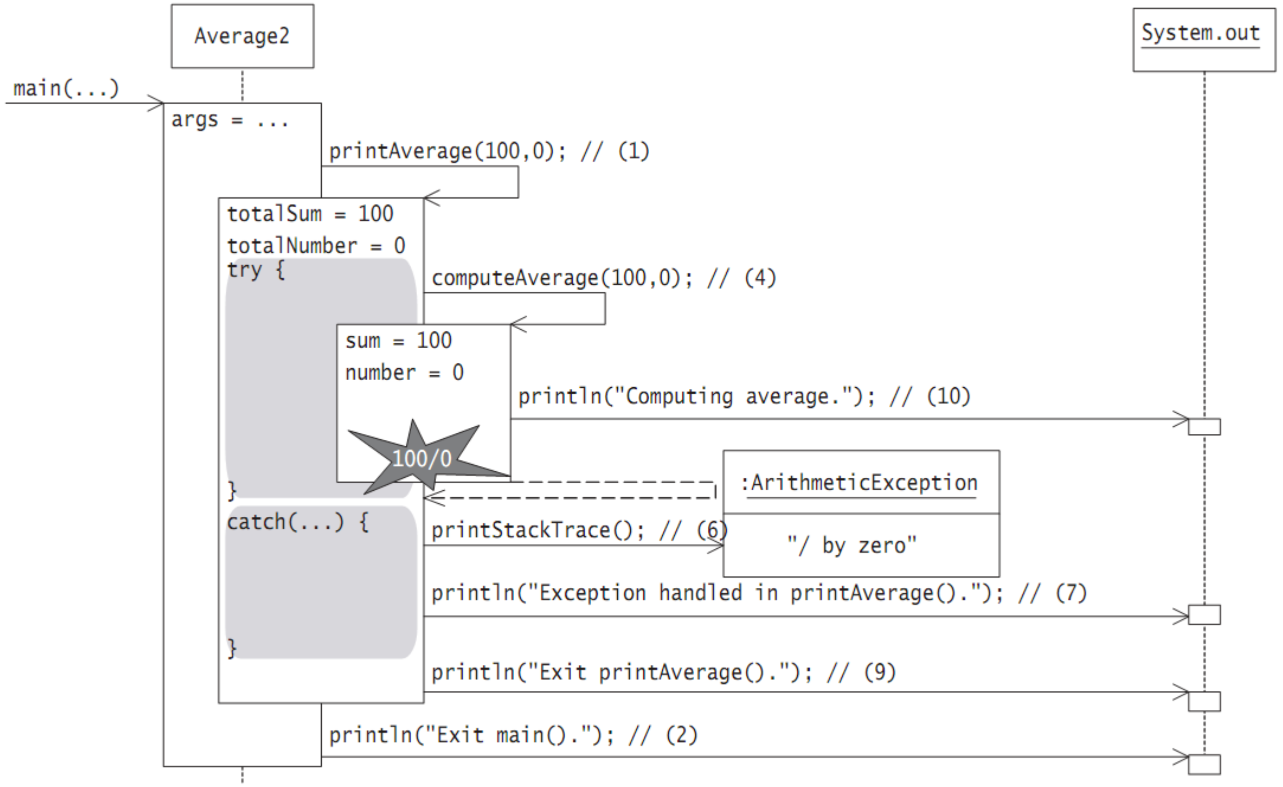
## Exception handling


Average2
System.out
main(...)
args = ...
printAverage(100,20); // (1)
totalSum = 100
totalNumber = 20
try {
computeAverage(100,20); // (4)
sum = 100
number = 20
println("Computing average."); // (10)
100/20
5 // (11)
average = 5
println("Average = 100 / 20 = 5"); // (5)
}
println("Exit printAverage()."); // (9)
println("Exit main()."); // (2)

Output from the program:

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

## Exception handling

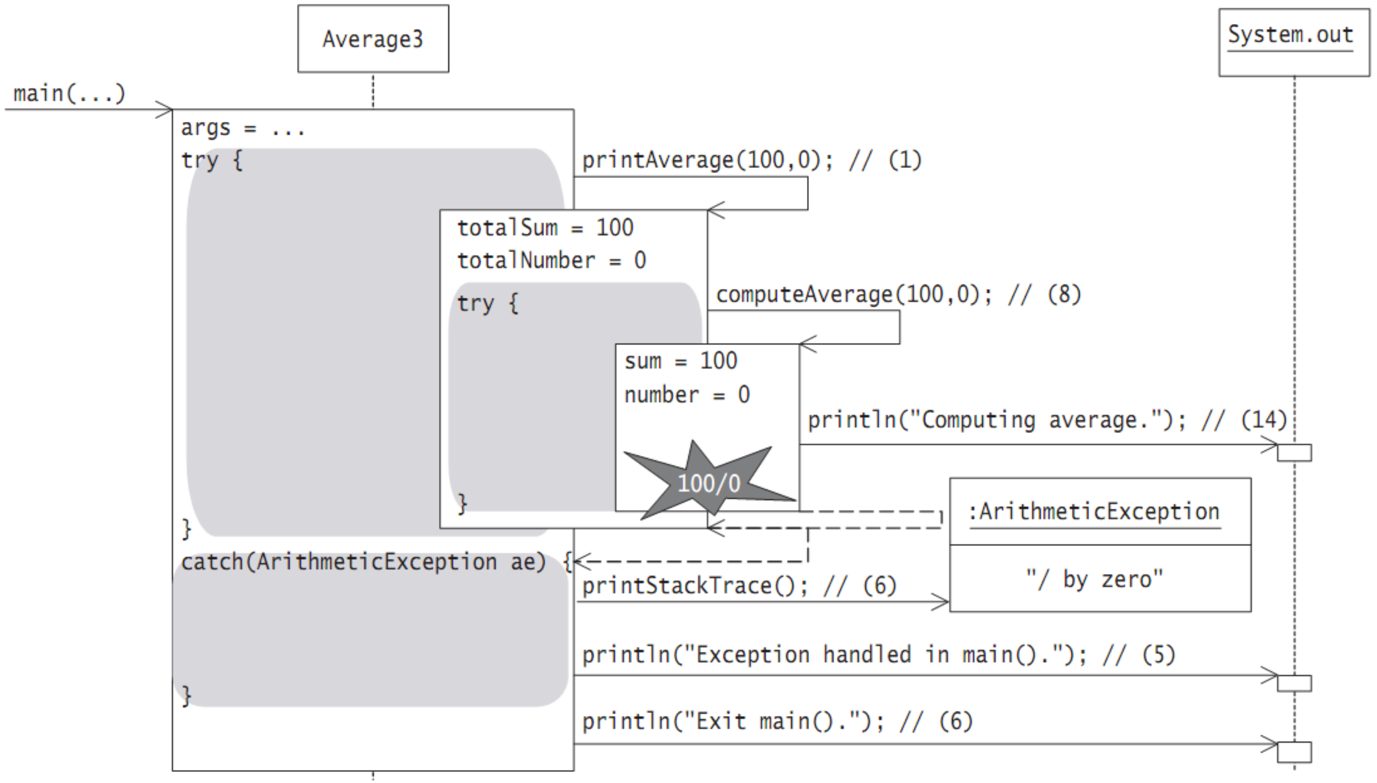

Average2
System.out
main(...)
args = ...
printAverage(100,0); // (1)
totalSum = 100
totalNumber = 0
try {
computeAverage(100,0); // (4)
sum = 100
number = 0
println("Computing average."); // (10)
100/0
:ArithmeticException
"/ by zero"
}
catch(...) {
printStackTrace(); // (6)
println("Exception handled in printAverage()."); // (7)
}
println("Exit printAverage()."); // (9)
println("Exit main()."); // (2)

Output from the program:

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average2.computeAverage(Average2.java:24)
        at Average2.printAverage(Average2.java:11)
        at Average2.main(Average2.java:5)
Exception handled in printAverage().
Exit printAverage().
Exit main().
```

```
public class Average3 {
  public static void main(String[] args) {
    try {                                                                // (1)
      printAverage(100, 0);                                              // (2)
    } catch (ArithmeticException ae) {                                   // (3)
      ae.printStackTrace();                                              // (4)
      System.out.println("Exception handled in main().");                // (5)
    }
    System.out.println("Exit main().");                                  // (6)
  }
public static void printAverage(int totalSum, int totalNumber) {
    try {                                                                // (7)
      int average = computeAverage(totalSum, totalNumber);               // (8)
      System.out.println("Average = " +                                  // (9)
          totalSum + " / " + totalNumber + " = " + average);
    } catch (IllegalArgumentException iae) {                             // (10)
      iae.printStackTrace();                                             // (11)
      System.out.println("Exception handled in printAverage().");        // (12)
    }
    System.out.println("Exit printAverage().");                          // (13)
  }
  public static int computeAverage(int sum, int number) {
    System.out.println("Computing average.");                            // (14)
    return sum/number;                                                   // (15)
  }
}
```

# Exception handling


Average3
System.out
main(...)
args = ...
try {
printAverage(100,0); // (1)
totalSum = 100
totalNumber = 0
try {
computeAverage(100,0); // (8)
sum = 100
number = 0
println("Computing average."); // (14)
100/0
:ArithmeticException
}
}
catch(ArithmeticException ae) {
printStackTrace(); // (6)
"/ by zero"
println("Exception handled in main()."); // (5)
}
println("Exit main()."); // (6)

Output from the program:

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average3.computeAverage(Average3.java:30)
        at Average3.printAverage(Average3.java:17)
        at Average3.main(Average3.java:6)
Exception handled in main().
Exit main().
```

## The finally Block

If the finally block neither throws an exception nor executes a control transfer statement like a return or a labeled break, the execution of the try block or any catch block determines how execution proceeds after the finally block

- If there is no exception thrown during execution of the try block or the exception has been handled in a catch block, normal execution continues after the finally block.
- If there is any pending exception that has been thrown and not handled (either due to the fact that no catch block was found or the catch block threw an exception), the method is aborted and the exception is propagated after the executionof the finally block.

# The try-catch-finally Construct

```
public class Average4 {
  public static void main(String[] args) {
    printAverage(100, 20);                                            // (1)
    System.out.println("Exit main().");                               // (2)
  }
  public static void printAverage(int totalSum, int totalNumber) {
    try {                                                             // (3)
      int average = computeAverage(totalSum, totalNumber);            // (4)
      System.out.println("Average = " +                               // (5)
          totalSum + " / " + totalNumber + " = " + average);
    } catch (ArithmeticException ae) {                                // (6)
      ae.printStackTrace();                                           // (7)
      System.out.println("Exception handled in printAverage().");     // (8)
    } finally {                                                       // (9)
      System.out.println("Finally done.");
    }
    System.out.println("Exit printAverage().");                       // (10)
  }
  public static int computeAverage(int sum, int number) {
    System.out.println("Computing average.");                         // (11)
    return sum/number;                                                // (12)
  }
}
```

## The **`throw`** Statement

Application can programmatically throw an exception using the throw statement.

The general format of the throw statement is as follows:

```
throw <object reference expression>;
```

## The **`throw`** Statement

The compiler ensures that the <object reference expression> is of the type Throwable class or one of its subclasses.

At runtime a NullPointerException is thrown by the JVM if the <object reference expression> is null.

This ensures that a Throwable will always be propagated.

A detail message is often passed to the constructor when the exception object is created.

```
throw new ArithmeticException("Integer division by 0");
```

## The `throw` Statement

- When an exception is thrown, normal execution is suspended. The runtime system proceeds to find a catch block that can handle the exception.
- The search starts in the context of the current try block, propagating to any enclosing try blocks and through the runtime stack to find a handler for the exception.
- Any associated finally block of a try block encountered along the search path is executed. If no handler is found, then the exception is dealt with by the default exception handler at the top level.
- If a handler is found, normal execution resumes after the code in its catch block has been executed, barring any rethrowing of an exception.

## The **`throws`** Clause

- A throws clause can be specified in the method header.

```
... someMethod(...) throws <ExceptionType1>,...,
                           <ExceptionTypen> { ... }
```

- Each <ExceptionType$_i$> declares an exception, normally only checked exceptions are declared. The compiler enforces that the checked exceptions thrown by a method are limited to those specified in its throws clause.
- Of course, the method can throw exceptions that are subclasses of the checked exceptions in the throws clause.
- The throws clause can specify unchecked exceptions, but this is seldom done and the compiler does not verify them.

## The **`throws`** Clause

If a checked exception is thrown in a method, it must be handled in one of three ways:

- By using a try block and catching the exception in a handler and dealing with it
- By using a  try block and catching the exception in a handler, but throwing another exception that is either unchecked or declared in its throws clause
- By explicitly allowing propagation of the exception to its caller by declaring it in the throws clause of its method header

## Assertions

- Each assertion contains a boolean expression that is expected to be true when the assertion is executed.
- If this assumption is false, the JVM throws a special error represented by the AssertionError class.
- The assertion facility uses the exception handling mechanism to propagate the error.
- Since the assertion error signals failure in program behavior, it should  not be caught programmatically, but allowed to propagate to the top level.

## Assertions

The assertion facility is an invaluable aid in implementing *correct* programs (i.e., programs that adhere to their specifications).

It should not be confused with the exception handling mechanism that aids in developing *robust* programs (i.e., programs that handle unexpected situations gracefully).

Used judiciously, the two mechanisms facilitate programs that are *reliable*.

## The assert Statement and the AssertionError Class

The following two forms of the assert statement can be used to specify assertions:

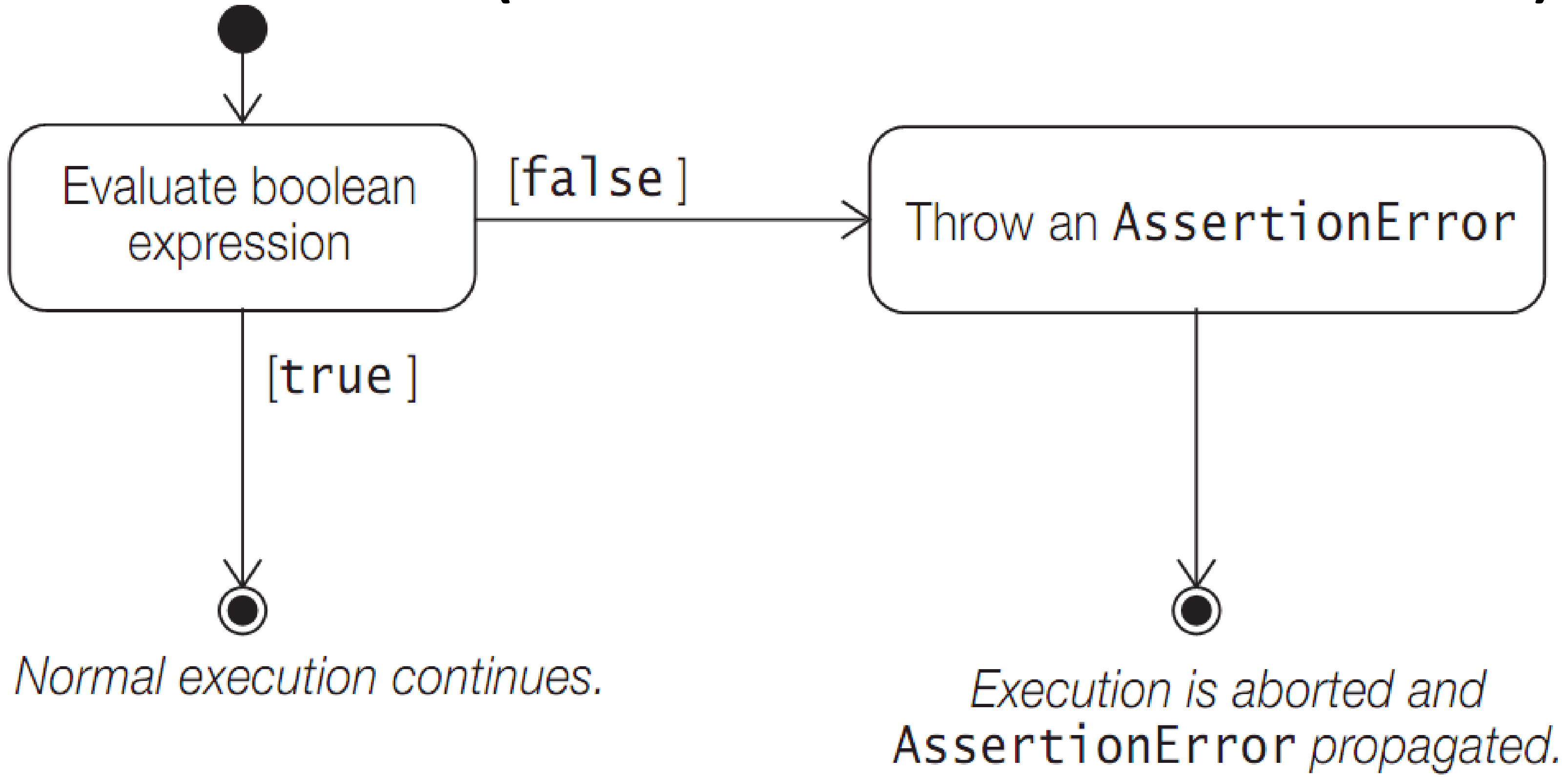
```
assert <boolean expression> ;    // the simple form
assert <boolean expression> : <message expression> ;
// the augmented form
```

# The assert Statement and the AssertionError Class

The two forms are essentially equivalent to the following code, respectively:

```
if (<assertions enabled> && !<boolean expression>)
// the simple form
   throw new AssertionError();
if (<assertions enabled> && !<boolean expression>)
// the augmented form
   throw new AssertionError(<message expression>);
```

# Execution of the Simple assert Statement (with Assertions Enabled)


Evaluate boolean expression
[false]
Throw an AssertionError
[true]
Normal execution continues.
Execution is aborted and AssertionError propagated.

## Compiling Assertions

The assertion facility was introduced in Java 1.4. Prior to Java 1.4, assert was an identifier and not a keyword.

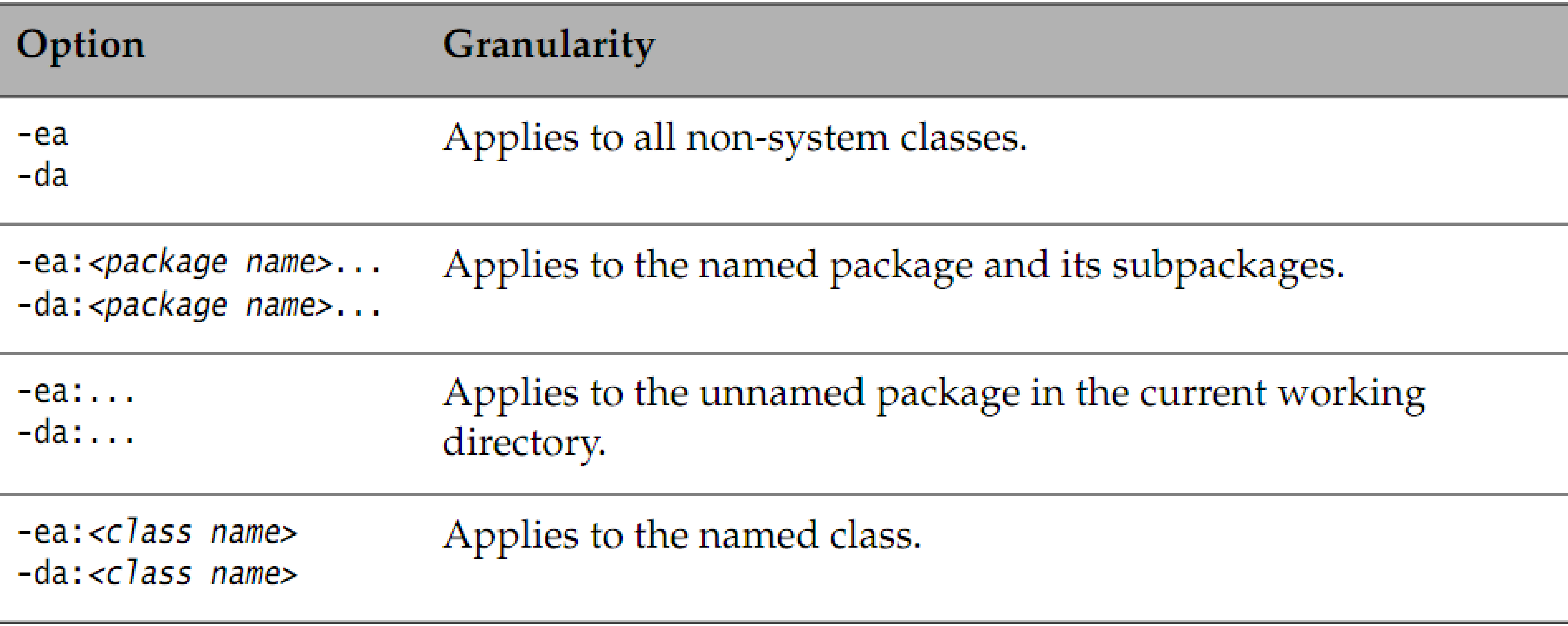
Starting with Java 1.4, it could only be used as a keyword in the source code. Also starting with Java 1.5, the javac compiler will compile assertions by default.

This means that incorrect use of the keyword assert will be flagged as a compile time error, e.g., if assert is used as an identifier.

## Runtime Enabling and Disabling of Assertions

- Two options are provided by the java command to enable and disable assertions with various granularities.
- The option **-enableassertions**, or its short form **-ea**, enables assertions, and the option **-disableassertions**, or its short form **-da**, disables assertions at various granularities.
- The granularities that can be specified are shown in Table

# Runtime Enabling and Disabling of Assertions



| Option | Granularity |
|---|---|
| `-ea`<br>`-da` | Applies to all non-system classes. |
| `-ea:`*`<package name>`*`...`<br>`-da:`*`<package name>`*`...` | Applies to the named package and its subpackages. |
| `-ea:...`<br>`-da:...` | Applies to the unnamed package in the current working directory. |
| `-ea:`*`<class name>`*<br>`-da:`*`<class name>`* | Applies to the named class. |

## Assertion Execution for All Non-System Classes

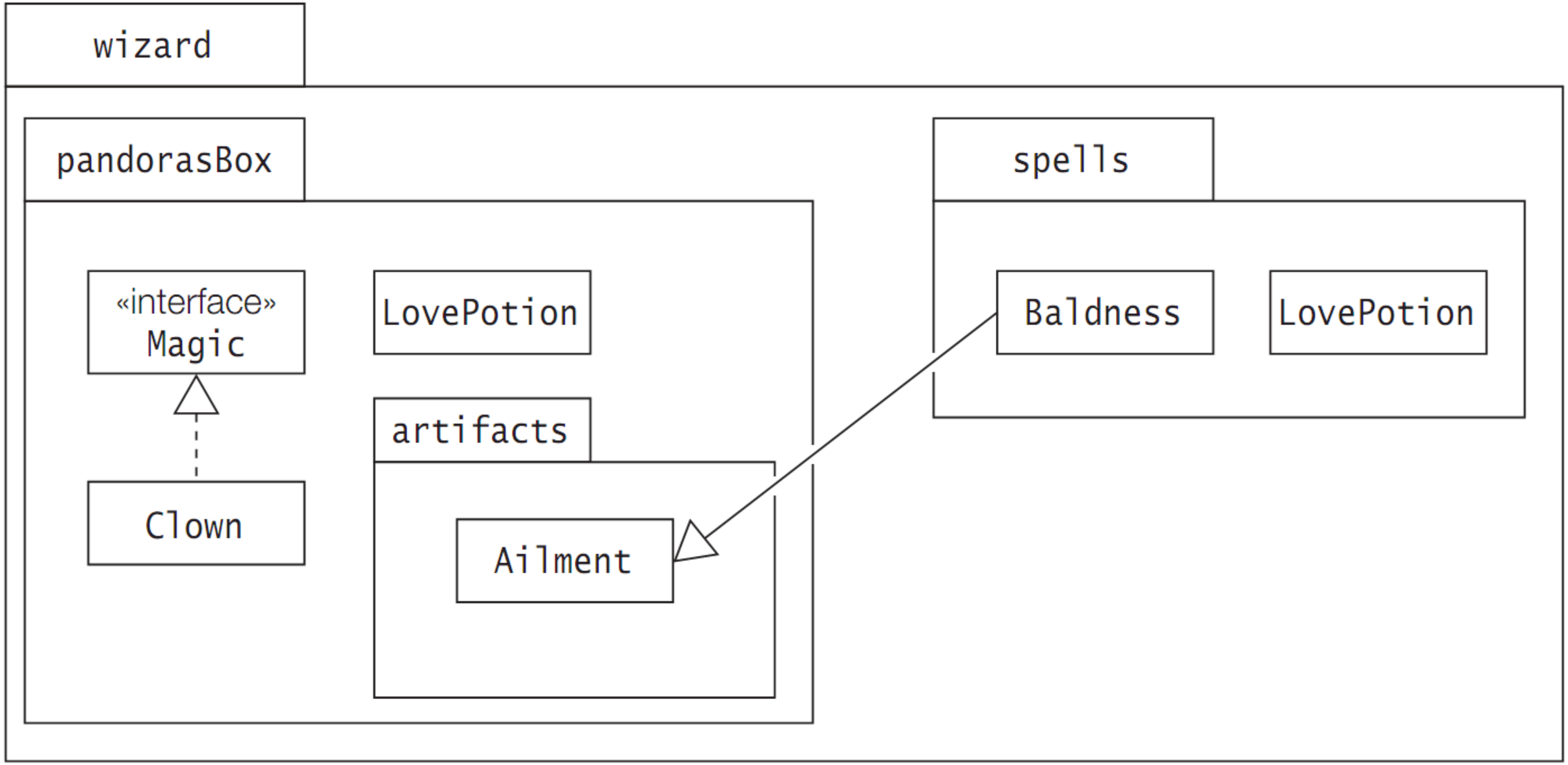
- The -ea option means that all *non-system classes* loaded during the execution of the program have their assertions enabled. A system class is a class that is in the Java platform libraries. For example, classes in the java.* packages are system classes.
- A system class is loaded directly by the JVM.
- Note that class files not compiled with an assertion-aware compiler are not affected, whether assertions are enabled or disabled.
- Also, once a class has been loaded and initialized at runtime, its assertion status cannot be changed.

## Assertion Execution at the Package Level

- Assume that we have a program called Trickster in the unnamed package, that uses the wizard package
- The following command line will only enable assertions for all classes in the package wizard.pandorasBox and its subpackage wizard.pandorasBox.artifacts.
- The assertions in the class Trickster are not enabled.

```
>java -ea:wizard.pandorasBox... Trickster
```

- Without the ... notation, the package name will be interpreted as a class name.
- Non-existent package names specified in the command line are silently accepted, but simply have no consequences during execution.


wizard
pandorasBox
«interface»
Magic
LovePotion
Clown
artifacts
Ailment
spells
Baldness
LovePotion

## Assertion Execution at the Class Level

The following command line will only enable assertions in the Trickster class.

```
>java -ea:Trickster Trickster
```

The following command line will only enable assertions in the specified class

wizard.pandorasBox.artifacts.Ailment,

and no other class.

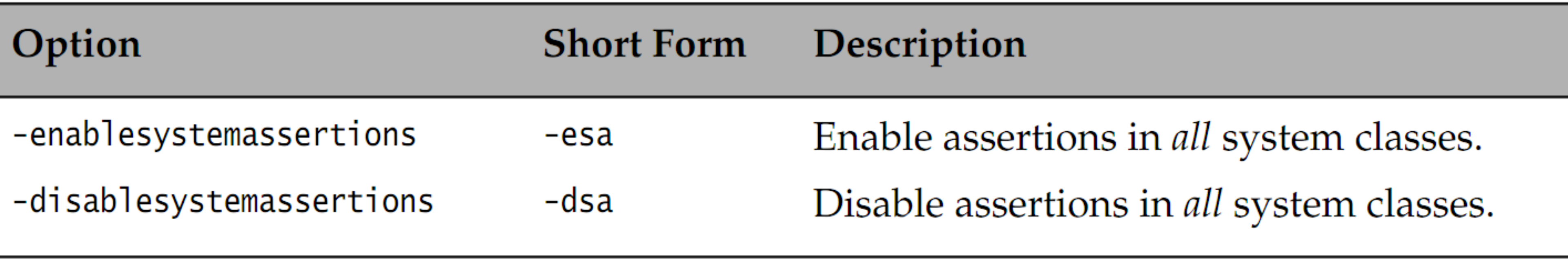
```
>java -ea:wizard.pandorasBox.artifacts.Ailment Trickster
```

# Assertion Execution for All System Classes

In the following command line, the first option -esa will enable assertions for all system classes. The second option -ea:wizard... will enable assertions in the package wizard and its subpackages wizard.pandorasBox, wizard.pandorasBox.artifacts and wizard.spells, but the third option -da:wizard.pandorasBox.artifacts... will disable them in the package wizard.pandorasBox.artifacts.

```
>java -esa -ea:wizard...
    -da:wizard.pandorasBox.artifacts... Trickster
```

# Enabling and Disabling Assertions in All System Classes at Runtime



| **Option** | **Short Form** | **Description** |
|---|---|---|
| `-enablesystemassertions` | `-esa` | Enable assertions in *all* system classes. |
| `-disablesystemassertions` | `-dsa` | Disable assertions in *all* system classes. |

## Using Assertions

- The assertion facility is a defensive mechanism, meaning that it should only be used to test the code, and should not be employed after the code is delivered.
- The program should exhibit the same behavior whether assertions are enabled or disabled.
- The program should not rely on any computations done within an assertion statement.

## Using Assertions

With assertions enabled, the following statement would be executed, but if assertions were disabled, it could have dire consequences.

```
assert reactor.controlCoreTemperature();
```

# Assertions: Internal Invariants

The following code at (1) makes the assumption that the variable status must be negative for the last else clause to be executed.

```
int status = ref1.compareTo(ref2);
if (status == 0) {
   ...
} else if (status > 0) {
   ...
} else {
// (1) status must be negative.
   ...
}
```

This assumption is an internal invariant and can be verified using an assertion, as

shown at (2) below.

```
int status = ref1.compareTo(ref2);
if (status == 0) {
   ...
} else if (status > 0) {
   ...
} else {
   assert status < 0 : status; // (2)
   ...
}
```

# Control Flow Invariants

The following idiom can be employed to explicitly test that certain locations in the code will never be reached.

```
assert false : "This line should never be reached.";
```

## Preconditions and Postconditions

The assertion facility can be used to practice a limited form of programming-by-contract. For example, the assertion facility can be used to check that methods comply with their contract.

*Preconditions* define assumptions for the proper execution of a method when it is invoked.

*Postconditions* define assumptions about the successful completion of a method.