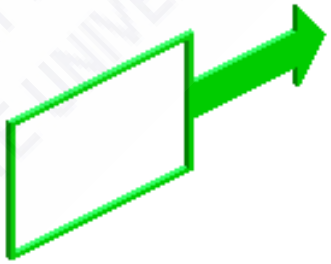
Introducing the Java (Cont.2)

Rules for Creating Statements

- Use a semicolon to terminate statements.
- Define multiple statements within braces.
- Use braces for control statements.





Control Statements

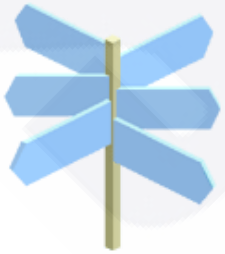
Categorizing Basic Flow Control Types

Flow control can be categorized into four types:

Sequential

Iteration

Selection



Transfer

Control Statements

Using Flow Control in Java

- Each simple statement terminates with a semicolon (;).
- Group statements by using the braces { }.
- Each block executes as a single statement within the flow of control structure.

```
{
  boolean finished = true;
  System.out.println("i = " + i);
  i++;
}
```

Creating Code Blocks

- Enclose all class declarations.
- Enclose all method declarations.
- Group other related code segments.

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

You never need to destroy an object

Scoping: { }

```
{
   int x = 12;
   /* only x available */
   {
      int q = 96;
      /* both x & q available */
   }
   /* only x available */
   /* q "out of scope" */
}
```

determines the **visibility** and **lifetime** of the names defined within that scope.

```
{
   int x = 12;
   {  ??????????????????
      int x = 96;   /* illegal */
   }
}
```

You never need to destroy an object

Scoping: { }

Scope of objects

```
{
    String s = new String("a string");
} /* end of scope */
```

vanishes at the end of the scope,

but the String object that s was pointing to is still occupying memory.

garbage collector

Control Statements

- **Three types of selection statements.**

1. `if` Single-Selection Statement

```
if ( studentGrade >= 60 )
   System.out.println( "Passed" );
```

2. `if…else` Double-Selection Statement

```
if ( grade >= 60 )
   System.out.println( "Passed" );
else
   System.out.println( "Failed" );
```

Can test multiple cases by placing `if…else` statements inside other `if…else` statements to create nested *`if…else`*

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else if ( studentGrade >= 80 )
   System.out.println( "B" );
else if ( studentGrade >= 70 )
   System.out.println( "C" );
else if ( studentGrade >= 60 )
   System.out.println( "D" );
else
   System.out.println( "F" );
```

Control Statements

- **Three types of selection statements. (Cont.1)**

2. `if…else` Double-Selection Statement (Cont.1)

Conditional operator (`?:`)—shorthand `if…else`.
Ternary operator (takes three operands)
Operands and `?:` form a conditional expression

Example:

```
System.out.println(
   studentGrade >= 60 ? "Passed" : "Failed" );
```

Control Statements

- **Three types of selection statements. (Cont.1)**

2. if…else Double-Selection Statement (Cont.2)

```
if ( x > 5 )
{
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
}
else
   System.out.println( "x is <= 5" );
-----------------------------------------------
if ( grade >= 60 )
   System.out.println("Passed");
else
{
   System.out.println("Failed");
   System.out.println("You must take this course again.");
}
```

Control Statements

- **Three types of selection statements. (Cont.1)**

3. `switch` statement

- Case labels must be constants.
- Use `break` to jump out of a switch.
- It is recommended to always provide a default.

```
switch (choice) {
   case 37:
      System.out.println("Coffee?");
      break;

   case 45:
      System.out.println("Tea?");
      break;

   default:
      System.out.println("???");
}
```

There are situations where falling through can be useful.

To fall through, simply do not include a break.

Control Statements

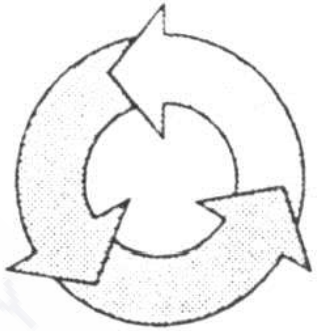
- **Three repetition statements (also called looping statements)**

  Repetition statement's body may be a single statement or a block { }

  - `while` **statement**
  - `for` **statement**
  - **The** `do...while`

Looping in Java

- There are three types of loops in Java:
  - `while`
  - `do...while`
  - `for`
- All loops have four parts:
  - Iteration condition
  - Body
  - Initialization
  - Termination



Using the `while` Loop

`while` is the simplest loop statement and contains the following general form:

```
while ( boolean_expr )
    statement;
```

Example:

```
int i = 0;
while (i < 10) {
    System.out.println("i = " + i);
    i++;
}
```

Using the do...while Loop

do...while loops place the test at the end:

```
do
    statement;
while ( termination );
```

Example:

```
int i = 0;
do {
  System.out.println("i = " + i);
  i++;
} while (i < 10);
```

Using the `for` Loop

`for` loops are the most common loops:

```
for ( initialization; termination; iteration )
    statement;
```

Example:

```
for (i = 0; i < 10; i++)
    System.out.println(i);
```

How would this `for` loop look using a `while` loop?

More About the `for` Loop

- Variables can be declared in the initialization part of a `for` loop:

```
for (int i = 0; i < 10; i++)
    System.out.println("i = " + i);
```

- Initialization and iteration can consist of a list of comma-separated expressions:

```
for (int i = 0, j = 10; i < j; i++, j--) {
    System.out.println("i = " + i);
    System.out.println("j = " + j);
}
```

Guided Practice: Spot the Mistakes

```
int x = 10;
while (x > 0);
   System.out.println(x--);
System.out.println("We have lift off!");
```

Extra semicolon

while (x > 0)

; // null loop body

```
int x = 10;
while (x > 0)
   System.out.println("x is " + x);
   x--;
```

2

Outside the loop

```
int sum = 0;
for (; i < 10; sum += i++);
System.out.println("Sum is " + sum);
```

i is not initialized anywhere

Form of the <u>infinite loop</u> is:

```
for(;;)

while(true)
```

break and continue

You can also control the flow of the loop inside the body of any of the iteration statements by using break and continue.

Break quits the loop without executing the rest of the statements in the loop.

Continue stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration.

```
package exh;
import java.util.*;

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

      int a = 1 ;

      for(int i = 0; i < 128; i++){

         if ( i == 120 )
           break;

        if ( i == 100 || i == 101 )
           continue;

         if (Character.isLowerCase((char)i))

            System.out.println("value: " + i + " character: " + (char)i);
      }
      System.out.println("----------------------------");

   }   }
```

```
value: 97 character: a
value: 98 character: b
value: 99 character: c
value: 102 character: f
value: 103 character: g
value: 104 character: h
value: 105 character: i
value: 106 character: j
value: 107 character: k
value: 108 character: l
value: 109 character: m
value: 110 character: n
value: 111 character: o
value: 112 character: p
value: 113 character: q
value: 114 character: r
value: 115 character: s
value: 116 character: t
value: 117 character: u
value: 118 character: v
value: 119 character: w
----------------------------
```

break and continue label

```
label1:
    outer-iteration {
        inner-iteration {
            //...
            break;                  // 1
            //...
            continue;               // 2
            //...
            continue label1;        // 3
            //...
            break label1;           // 4
        }
    }
```

the break breaks out of the inner iteration and you end up in the outer iteration.

break and continue label

```
label1:
   outer-iteration {
      inner-iteration {
         //...
         break;                  // 1
         //...
         continue;               // 2
         //...
         continue label1;        // 3
         //...
         break label1;           // 4
      }
   }
```

the continue moves back to the beginning of the inner iteration.

break and continue label

```
label1:
   outer-iteration {
      inner-iteration {
         //...
         break;                  // 1
         //...
         continue;               // 2
         //...
         continue label1;        // 3
         //...
         break label1;           // 4
      }
   }
```

the continue label1 breaks out of the inner iteration *and* the outer iteration, all the way back to label1. Then it does in fact continue the iteration, but starting at the outer iteration.

break and continue label

```
label1:
   outer-iteration {
      inner-iteration {
         //...
         break;                  // 1
         //...
         continue;               // 2
         //...
         continue label1;        // 3
         //...
         break label1;           // 4
      }
   }
```

the break label1 also breaks all the way out to label1, but it does not reenter the iteration. It actually does break out of both iterations.

```
public class BM {
  public static void main(String[] args) {
    int i = 0;

    outer: // Can't have statements here

    for(; true ;) { // infinite loop

      inner: // Can't have statements here

      for(; i < 10; i++) {
        System.out.println("i = " + i);

        if(i == 2) {System.out.println("continue"); continue;}

        if(i == 3) {System.out.println("break");
          i++; // Otherwise i never gets incremented.
          break; }

        if(i == 7) {System.out.println("continue outer");
          i++; // Otherwise i never gets incremented.
          continue outer; }

        if(i == 8) {System.out.println("break outer"); break outer;}

        for(int k = 0; k < 5; k++) {

          if(k == 3) {System.out.println("continue inner");
            continue inner;} }
 }        }      }       }
```

```
i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
```

Implementing the `break` Statement

- Breaks out of a loop or switch statement
- Transfers control to the first statement after the loop body or `switch` statement
- Can simplify code but should be used sparingly

```
...
while (age <= 65) {
    balance = (balance+payment) * (1 + interest);
    if (balance >= 250000)
        break;
    age++;
}
...
```

Comparing Labeled `break` and `continue` Statements

Can be used to break out of nested loops or continue a loop outside the current loop:

```
outer_loop:
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 5; j++) {
        System.out.println(i);
        System.out.println(j);
        if (i + j > 7)
            break outer_loop;
    }
}
...
```

Compound Assignment Operators

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Fig. 4.13 | Arithmetic compound assignment operators.

Increment and Decrement Operators

Operator	Operator name	Sample expression	Explanation
++	prefix increment	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	postfix increment	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	prefix decrement	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	postfix decrement	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

Fig. 4.14 | Increment and decrement operators.

Logical Operators

- **Java's logical operators enable you to form more complex conditions by combining simple conditions.**
- **The logical operators are**
  - **&&** (conditional AND)
  - **||** (conditional OR)
  - **&** (boolean logical AND)
  - **|** (boolean logical inclusive OR)
  - **^** (boolean logical exclusive OR)
  - **!** (logical NOT).
- **[*Note:* The &, | and ^ operators are also bitwise operators when they are applied to integral operands.]**

Operators	Associativity	Type
++ --	right to left	unary postfix
++ -- + - ! (*type*)	right to left	unary prefix
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	boolean logical AND
^	left to right	boolean logical exclusive OR
|	left to right	boolean logical inclusive OR
&&	left to right	conditional AND
||	left to right	conditional OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Fig. 5.19 | Precedence/associativity of the operators discussed so far.

UML activity diagrams to summarize Java's control statements.

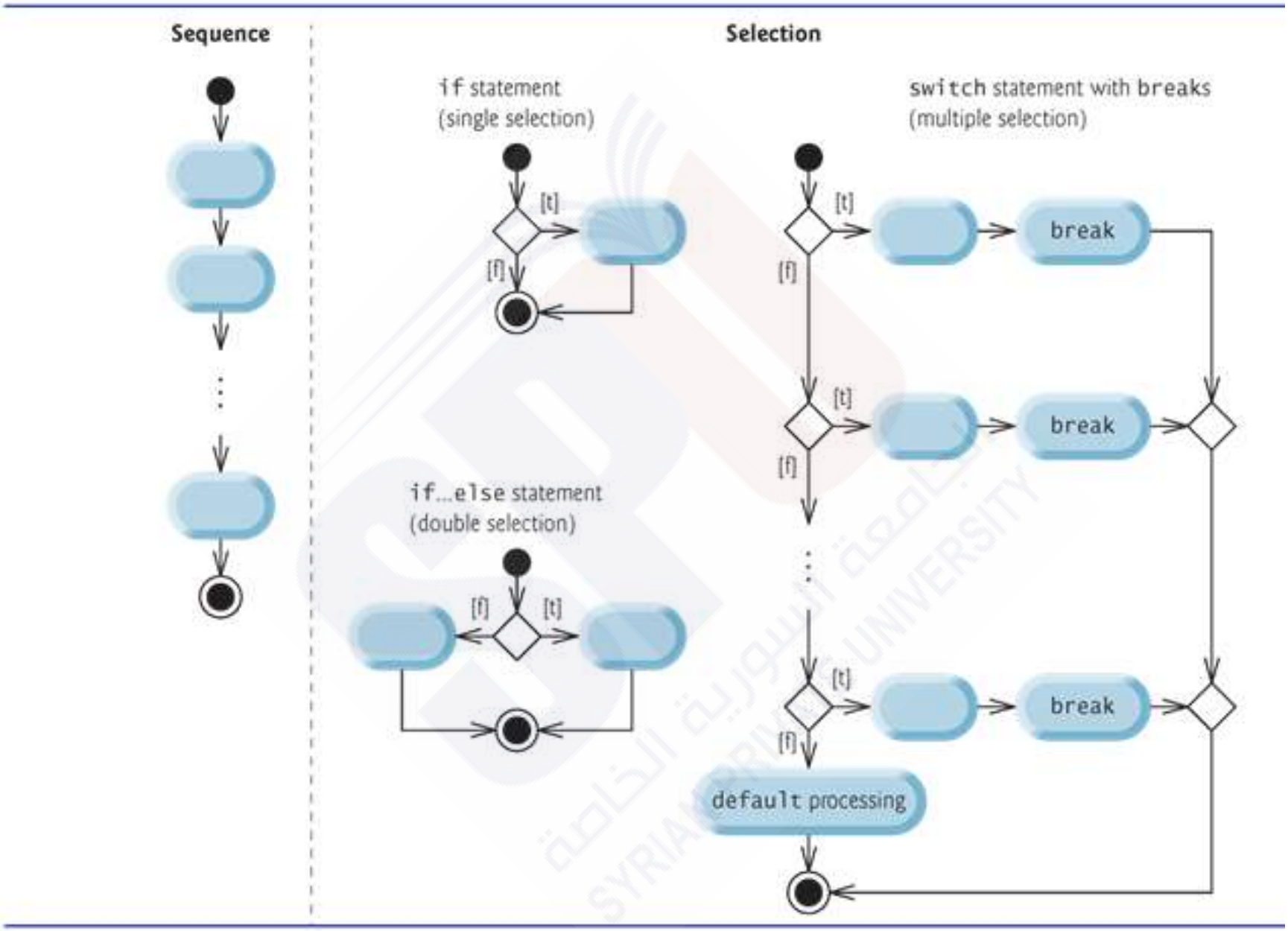


Fig. 5.20 | Java's single-entry/single-exit sequence, selection and repetition statements. (Part 1 of 2.)

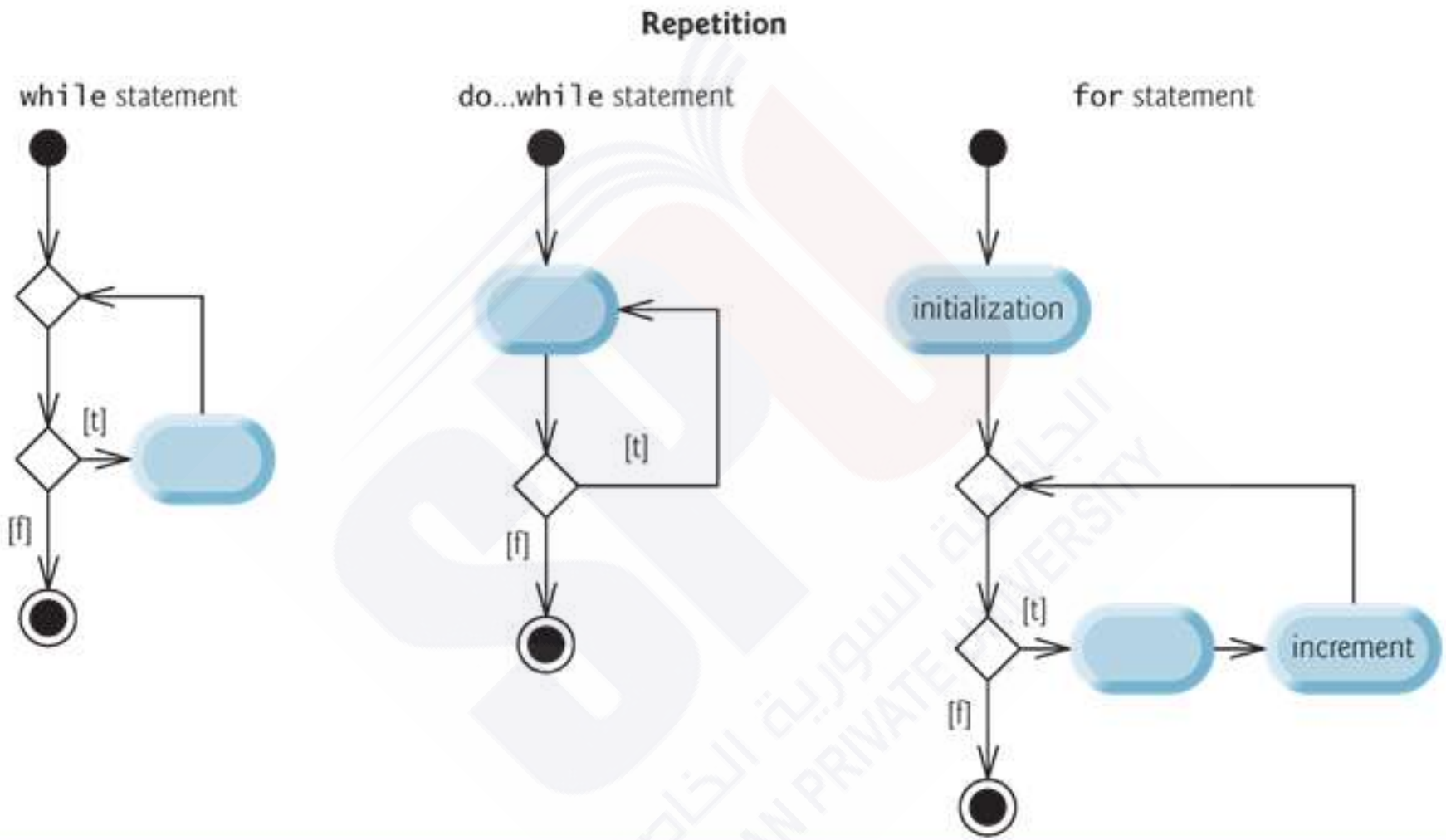


Fig. 5.20 | Java's single-entry/single-exit sequence, selection and repetition statements. (Part 2 of 2.)

Fig. 5.22 | Simplest activity diagram.

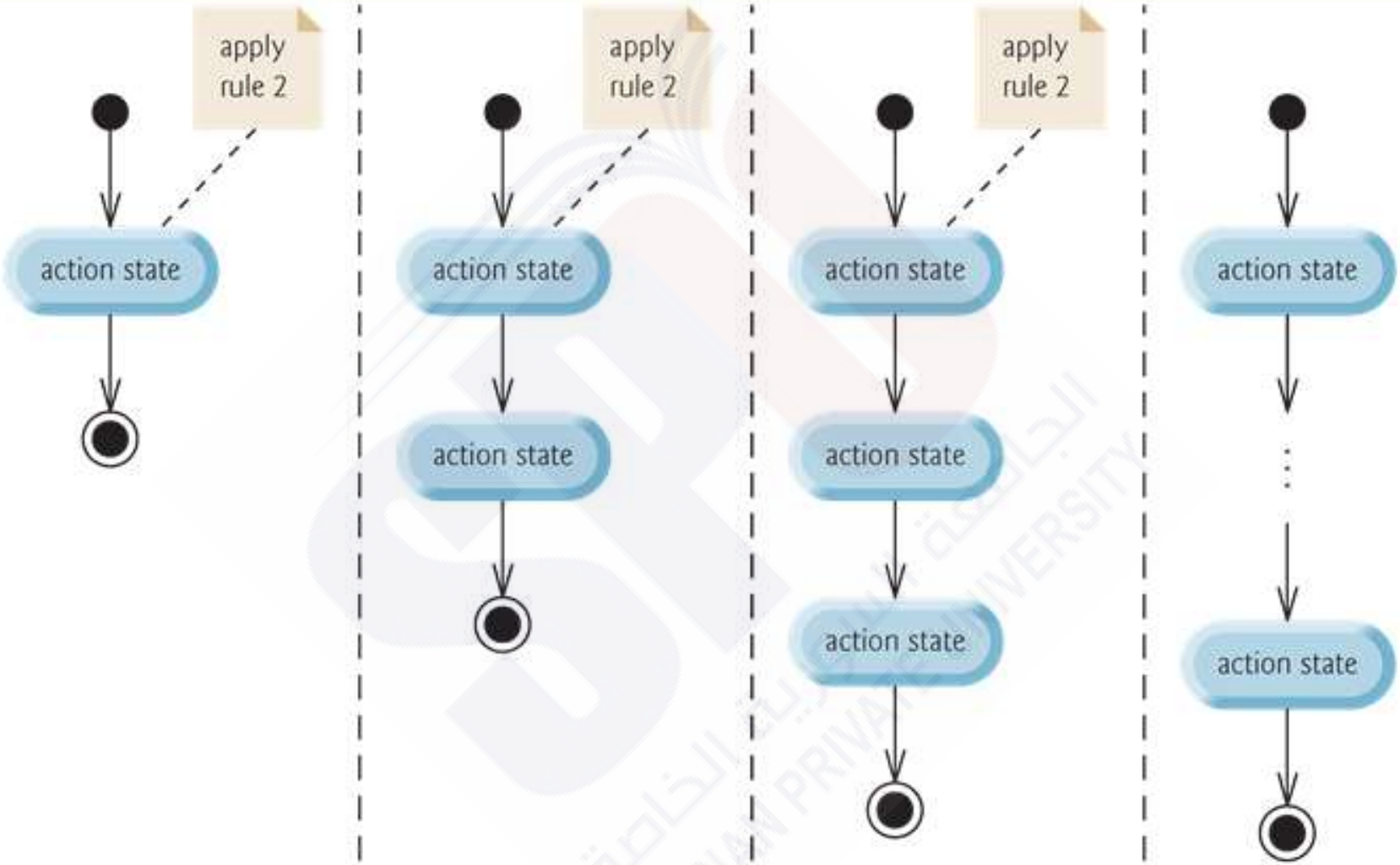


Fig. 5.23 | Repeatedly applying the stacking rule (rule 2) of Fig. 5.21 to the simplest activity diagram.

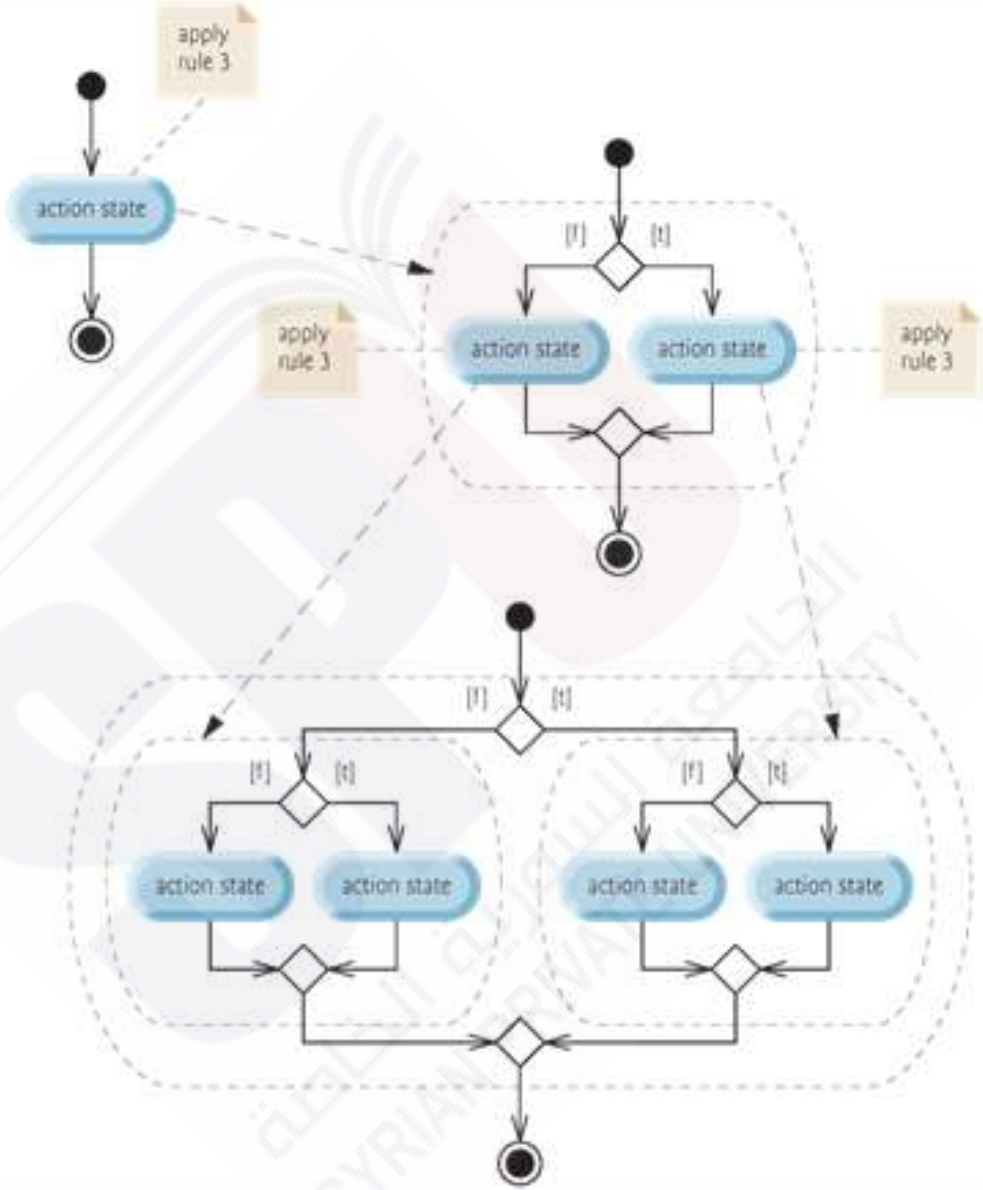


Fig. 5.24 | Repeatedly applying the nesting rule (rule 3) of Fig. 5.21 to the simplest activity diagram.

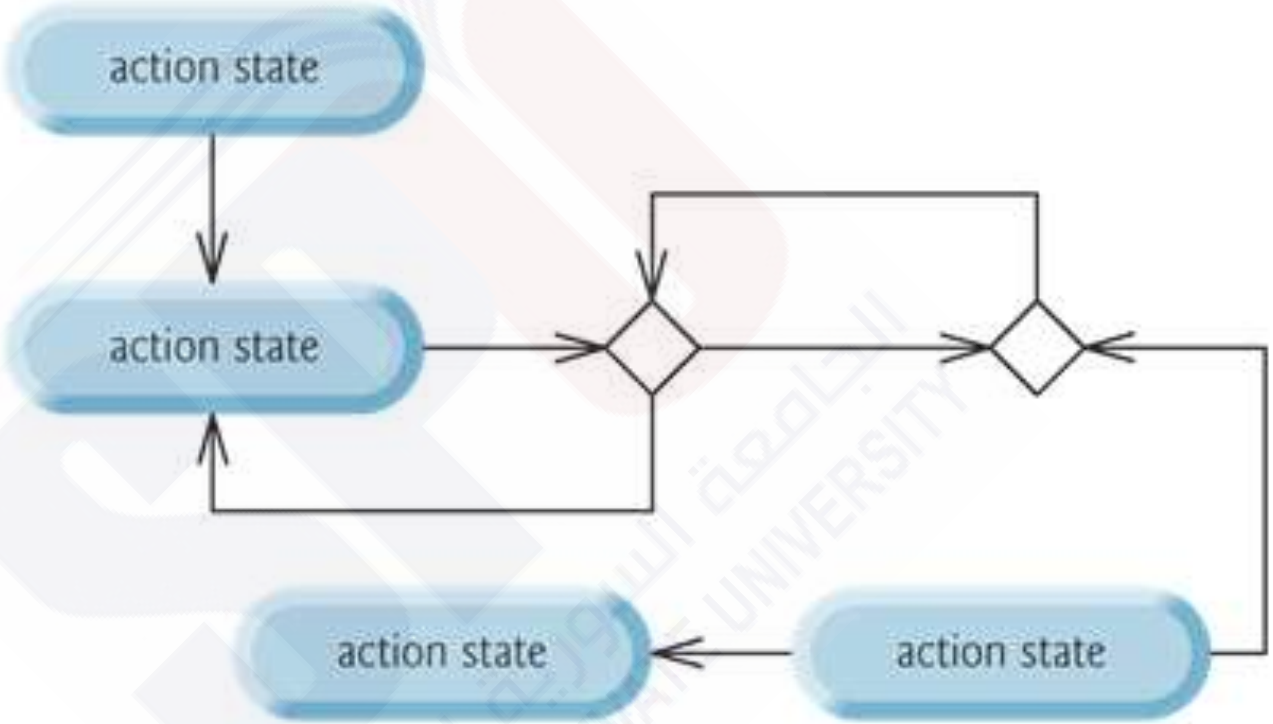


Fig. 5.25 | "Unstructured" activity diagram.