Chapter 3

Making Decisions

Creating Decision-Making Statements

The if/else statement

```
if (booleanExpression) {
   //Branch if true
} else {
   //Branch if false
}
```

Shortening Code with Pattern Matching

```
void compareIntegers(Number number){
  if(number instanceof Integer data){
    System.out.print(data.compareTo(5));
  }
}
```

"data" is referred to as the pattern variable & it can be reassigned (BAD practice)

can prevent the reassignment with a final modifier

```
if(number instanceof final Integer data){
  data = 10; // Error
}
```

We can use the pattern variable in an expression in the same line in which it is declared:

```
void printIntegerGreaterThan5(Number number) {
  if (number instanceof Integer data &&
        data.compareTo(5) > 0)
    System.out.println(data);
}
```

The type of the pattern variable must be a subtype of the variable on the left side of the expression:

```
Integer value = 123;
if (value instanceof Integer) {}
if (value instanceof Integer data) {} // Error
```

Limitations of Subtype Enforcement:

```
Number value = 123;
if (value instanceof List data) {}
```

List → interface

Flow Scoping

```
void printIntOrNumGreaterThan5(Number num){
  if(num instanceof Integer data ||
      data.compareTo(5)>0)
    System.out.print(data);
}
```

If the input doesn't inherit Integer, the "data" variable is undefined

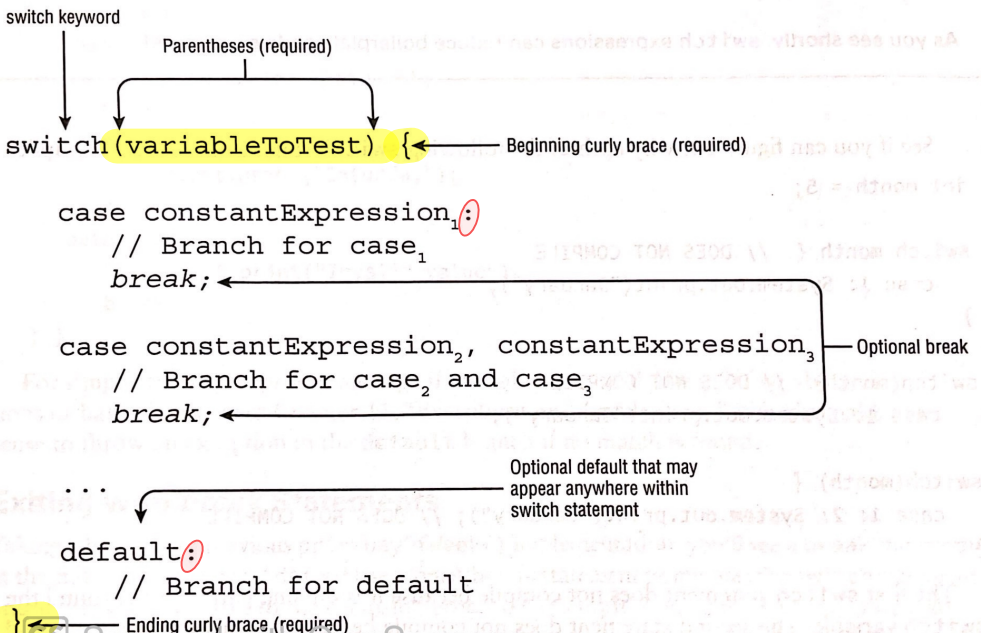
|| → CANNOT guarantee that "data" is an instance of Integer → data is not in scope → Error

```
void printOnlyIntegers(Number num){
  if(!(num instanceof Integer data))
    return;
  System.out.print(data.intValue());
}
```

When the last line of the method is reached, the input must inherit Integer, & therefore "data" stays in scope even after the if statement ends

Applying switch Statements

The switch Statement



`switch (month) { }` is perfectly valid

Without break statements, the code will execute every branch following a matching case statement, including any default statements it finds

```
public void printSeason(int month) {
    switch (month) {
        case 1, 2, 3: System.out.print("Winter");
        case 4, 5, 6 : System.out.print("Spring");
        default:       System.out.print("Unknown");
        case 7, 8, 9:  System.out.print("Summer");
        case 10, 11, 12: System.out.print("Fall");
    }}
```

printSeason(2) prints everything

Selecting switch Data Types

int + Integer

byte + Byte

short + Short

char + Character

String

enum values

var (if the type resolves to one of the preceding types)

Determining Acceptable Case Values

literals

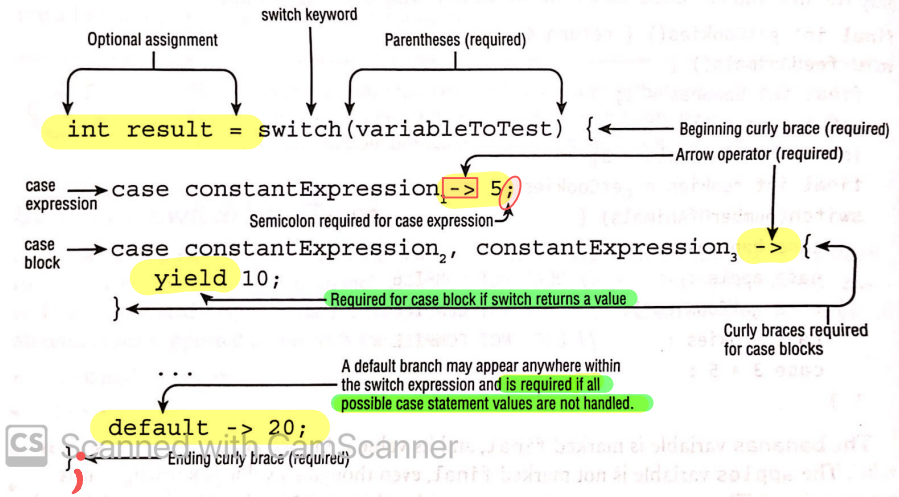
enum constants

final const. variables

→ same data type as the switch value

```
final int getCookies() { return 4; }
void feedAnimals() {
  final int bananas = 1;
  int apples = 2;
  int numberOfAnimals = 3;
  final int cookies = getCookies();
  switch (numberOfAnimals) {
    case bananas:
    case apples: // Error, not final
    case getCookies(): // Error, methods are not
    case cookies: // evaluated until runtime
    case 3*5:
  }}
```

The switch Expression




```
public void printSeason(int month) {
    switch (month) {
        case 1,2,3 -> System.out.print("Winter");
        case 4,5,6 -> System.out.print("Spring");
        case 7,8,9 -> System.out.print("Summer");
        case 10,11,12 -> System.out.print("Fall");
    }}
```

Don't need break, since only one branch is executed

3 new rules

#1 Returning Consistent Data Types

```
int measurement = 10;
int size = switch(measurement) {
    case 5 -> 1;
    case 10 -> (short) 2;
    default -> 5;
    case 20 -> "3"; // Error
    case 40 -> 4L; // Error
    case 50 -> null; // Error
};
```

#2 Applying a case Block

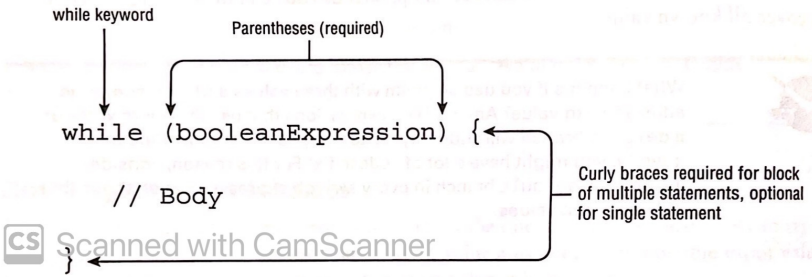
```
int fish = 5;
int length = 12;
var name = switch(fish) {
   case 1 -> "Goldfish";
   case 2 -> { } // Error
   case 3 -> {
      if(length > 10) yield "Blobfish";
   } // Error
   default -> "Swordfish";
};
```

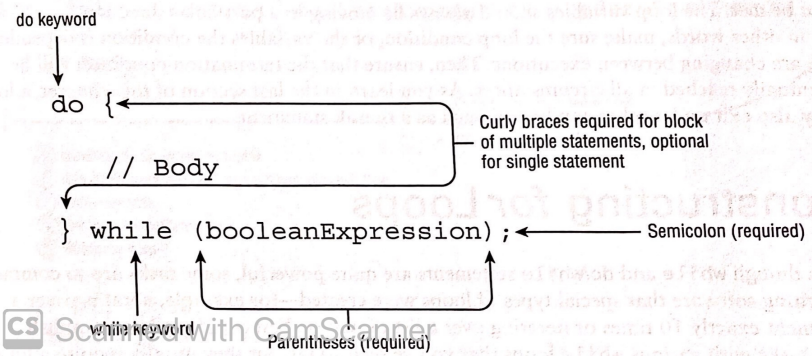
#3 Covering All Possible Values

```
enum Season { WINTER, SPRING,
              SUMMER, FALL }

String getWeather(Season value) {
  return switch(value) {
    case WINTER -> "Cold";
    case SPRING -> "Rainy";
    case SUMMER -> "Hot";
    case FALL -> "Warm";
  };
}
```

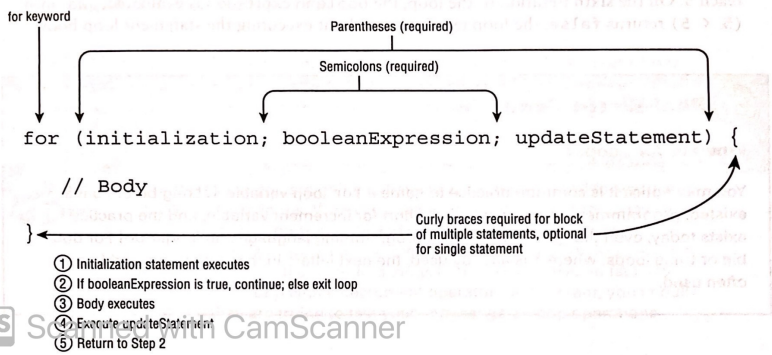
Writing while Loops





Constructing for Loops

The for Loop



Working with for Loops

#1 Creating an Infinite Loop

```
for(;;)
    System.out.println("Hello World");
```

#2 Adding Multiple Terms to the for Statement

```
int x = 0;
for (long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
  System.out.print(y + " "); }
System.out.print(x + " ");
```

#3 Redeclaring a Variable in the Initialization Block

```
int x = 0;
for (int x = 4; x < 5; x++)
  System.out.print(x + " ");
```

#4 Using Incompatible Data Types in the Initialization Block

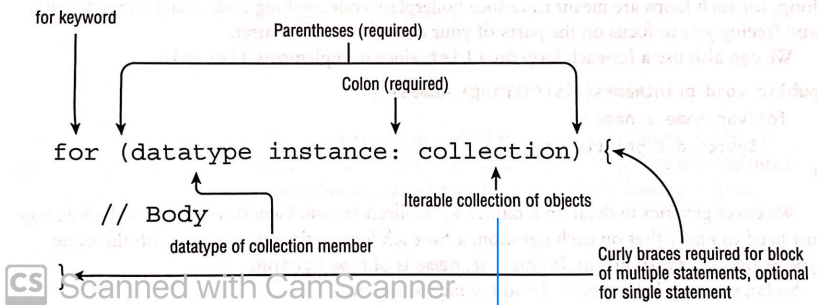
```
int x=0;
for(long y=0, int z=4; x<5; x++)
  System.out.print(y+" ");
```

must all be of **THE SAME** type

#5 Using Loop Variables Outside the Loop

```
for(long y=0, x=4; x<5 && y<10; x++, y++)
  System.out.print(y+" ");
System.out.print(x);
```

The for-each Loop



must be one of the following:

- A built-in Java array
- An object whose type implements java.lang.Iterable (List, Set)

```
public void printNames(String[] names) {
    for (var name : names)
        System.out.println(name);
}
```

Controlling Flow with Branching

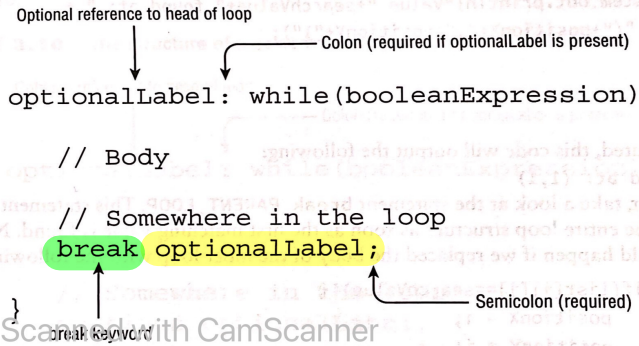
Adding Optional Labels

```
int[][] myComplexArray = {{1,2},{3,4}};
OUTER_LOOP: for (int[] arr : myComplexArray) {
  INNER_LOOP: for (int i=0; i<arr.length; i++){
    System.out.print(arr[i]+"\t");
  }
  System.out.println();
}
```

It is possible to add optional labels to control & block statements

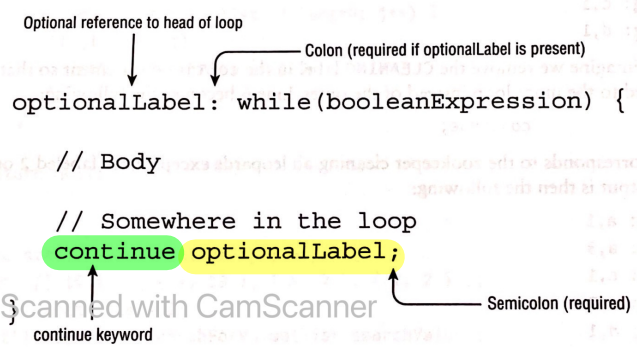
```
int frog = 15;
BAD_IDEA: if (frog > 10)
EVEN_WORSE_IDEA: { frog++; }
```

The break Statement



Without a label parameter, the break statement will terminate the nearest inner loop it is currently in the process of executing

The continue Statement



Ends the current iteration of the loop

Unreachable Code

```
int checkDate = 0;
while (checkDate < 10) {
  checkDate++;
  if (checkDate > 100) {
    break;
    checkDate++; // Error
  }
}
```

The compiler doesn't allow any statements to immediately follow a break, continue, or return statement

Reviewing Branching

	Support labels	Support break	Support continue	Support yield
while	✓	✓	✓	✗
do/while	✓	✓	✓	✗
for	✓	✓	✓	✗
switch	✓	✓	✗	✓

Review Questions

1/ A B C E F G ✓

2/ B ✓

3/ A D F **H** (List, Set)

4/ F ✓

5/ ~~G~~ **E** (int & Object is fine)

6/ C D E ~~F~~ (1 if - 1 else)

7/ B D ✓ (list data type only once in for loop)

8/ G ✓

9/ B C E ✓

10/ E (15, 16, 18, 19) ✓

11/ A ✓

12/ C ✓

13/ G ✓ (missing parentheses)

14/ B D F ✓

15/ F ✓

16/ A B D ✓

17/ B E ✓

18/ C E ~~F~~ (else ≠ if)

19/ E ✓

20/ — **A E**

21/ E (21, 23, 24, 26) ✓

22/ E ✓

23/ F ✓

24/ ~~D~~ **G** (no "in" in Java)

25/ D ✓

26/ F ✓

27/ F ✓

28/ ~~B~~ **F** (guppy is in scope after L.41-42)

29/ C ✓

75,9%