Control Flow

2020 Spring: AP Computer Science A

January 8th, 2020

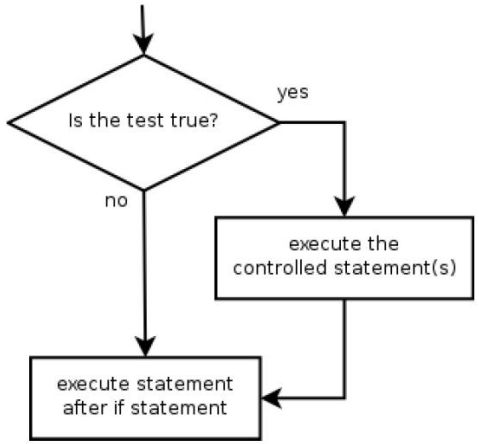
Today

- **Conditional execution**
  - if statement
  - switch statement
- **Loops**
  - while loops
  - do-while loops
  - for loops
  - break, continue
- **Methods**
- **Exceptions**
- **Formatting Text**

`if` Statement

- **Executes a block of statements only if a test is true**
  - Test should be evaluated to either true or false

```
if (test) {
    statement;
    ...
    statement;
}
```



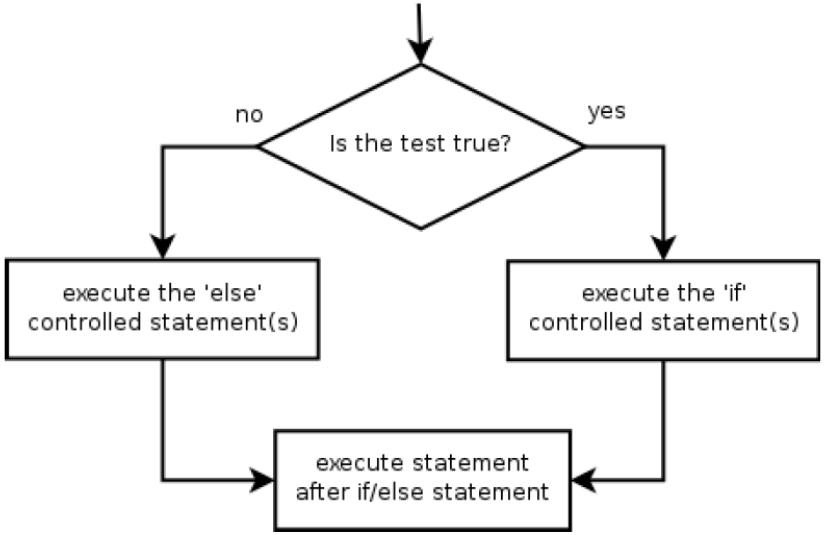
- **Example**

```
int x = 5;
if (x > 3) {
    System.out.println("x is greater than 3");
}
```

`if/else` Statement

- **Executes if block if a test is true, executes else block otherwise**
  - Only one of the statements will be executed!

```
if (test) {
    statement(s);
} else {
    statement(s);
}
```



- **Example**

```
int x = 5;
if (x > 3) {
    System.out.println("x is greater than 3");
} else {
    System.out.println("x is not greater than 3");
}
```

Misuse of `if`

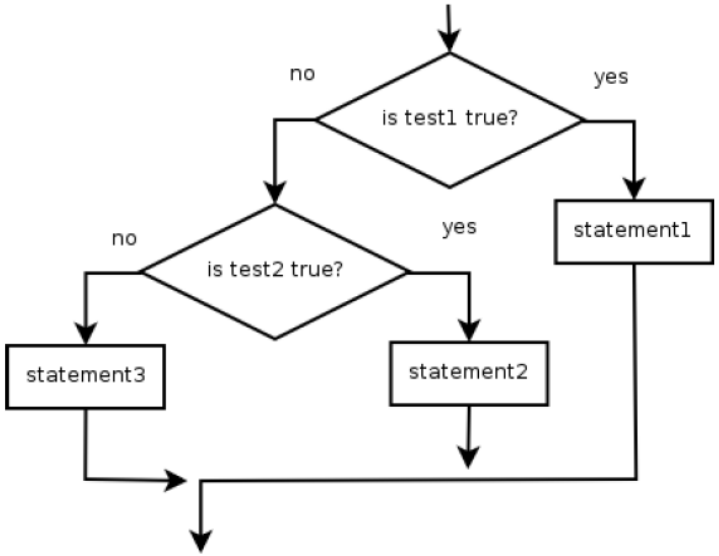
- **What's wrong with this?**

```
Scanner sc = new Scanner(System.in);
System.out.print("What percentage did you earn? ");
int percent = sc.nextInt();
if (percent >= 90) {
    System.out.println("You got an A!");
}
if (percent >= 80) {
    System.out.println("You got a B!");
}
if (percent >= 70) {
    System.out.println("You got a C!");
}
if (percent >= 60) {
    System.out.println("You got a D!");
}
if (percent < 60) {
    System.out.println("You got an F!");
}
```

else if Statement

- **Chooses between outcomes using many tests**

```
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else {
    statement(s);
}
```



- **Example**

```
double y = 0;
if (y > 0) {
    System.out.println("positive");
} else if (y < 0) {
    System.out.println("negative");
} else {
    System.out.println("zero");
}
```

`else if` Statement

- If it ends with `else`, *exactly one path* must be taken
- If it ends with `if`, the code *might not execute* any path

- Example

```
if (place == 1) {
    System.out.println("Gold!");
} else if (place == 2) {
    System.out.println("Silver!");
} else if (place == 3) {
    System.out.println("Bronze!");
}
```

`if/else` Structures

- **Exactly 1 path**
  - Mutually exclusive

```
if (test) {
     statement(s);
} else {
     statement(s);
}
```

- **0 or 1 path**
  - Mutually exclusive

```
if (test) {
     statement(s);
} else if (test) {
     statement(s);
} else if {
     statement(s);
}
```

- **0, 1, or many paths**
  - Independent tests
  - Not exclusive

```
if (test) {
     statement(s);
}
if (test) {
     statement(s);
}
if (test) {
     statement(s);
}
```

Exercises

- #1330 두 수 비교하기

- #9498 시험 성적

Nested `if`

- **`if` can contain `if` statements**

```
int num1 = 52, num2 = 32, num3 = 1;
if (num1 > num2) {
    if (num1 > num3) {
        System.out.println("num1 is the largest");
    }
}
```

- Try changing the condition above to a **single** `if` statement
  - *Hint: Use boolean operators!*

Dangling `else`

- **How should we interpret this code?**

```
int num1 = 152, num2 = 173;
if (num1 > num2)
    if (num1 > 100)
        System.out.println("num1 = " + num1);
else
    if (num2 > 100)
        System.out.println("num2 = " + num2);
System.out.println("Done.");
```

Dangling `else`

- **Which `if` statement should `else` be paired with?**

```
int num1 = 152, num2 = 173;
if (num1 > num2)
    if (num1 > 100)
        System.out.println("num1 = " + num1);
else
    if (num2 > 100)
        System.out.println("num2 = " + num2);
System.out.println("Done.");
```

- **Dangling `else` will be paired with the *nearest* `if`**

Dangling `else`

- **Should be fixed this way**

```
int num1 = 152, num2 = 173;
if (num1 > num2) {
    if (num1 > 100)
        System.out.println("num1 = " + num1);
} else {
    if (num2 > 100)
        System.out.println("num2 = " + num2);
}
System.out.println("Done.");
```

- **Use {} to explicitly mark the boundaries of `if/else` statements**
  - The code inside {} is called a ***block***

Exercises

- #10817 세 수

- #2753 윤년

`switch` Statement

- **`expression` is evaluated to an *integral value***
- **If that value equals any of val1, val2, ...**
  - The statements inside the corresponding value will be executed
  - And keeps executing the next statement until `break` is found
  - If corresponding value doesn't exist, statements in `default` is executed
  - `default` can be omitted

```
switch (expression) {
case val1:
    statement(s);
    break;
case val2:
    statement(s);
    break;
...
default:
    statement(s);
    break;
}
```

switch Statement Example

- **What is the output?**

```
int num = 2;
switch (num) {
case 1:
    System.out.println("Good morning, Java!");
    break;
case 2:
    System.out.println("Good afternoon, Java!");
    break;
case 3:
    System.out.println("Good evening, Java!");
    break;
default:
    System.out.println("Hello, Java!");
    break;
}
```

`switch` Statement Example

- **What is the output?** ***(Look out for*** **`break`** ***s)***

```
int num = 2;
switch (num) {
case 1:
    System.out.println("Good morning, Java!");
    break;
case 2:
    System.out.println("Good afternoon, Java!");
case 3:
    System.out.println("Good evening, Java!");
default:
    System.out.println("Hello, Java!");
    break;
}
```

Exercise

- You are given an integer. Use the switch statement to determine the remainder of that integer, when divided by 4.
- The output of your program should look like this.

```
Enter an integer: 9
The remainder is 1
```

```
Enter an integer: 10
The remainder is 2
```

```
Enter an integer: 11
The remainder is 3
```

```
Enter an integer: 12
The number is a multiple of 4
```

Loops

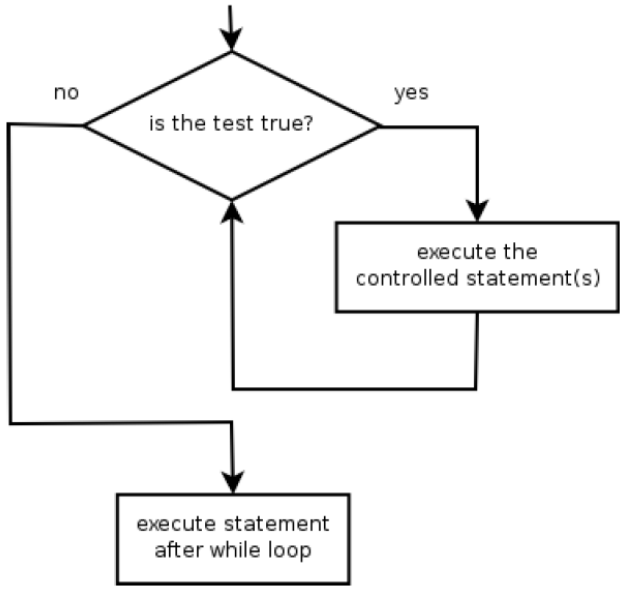
- **Definite loop:** Executes a *known number of times*
  - `for` loops are definite loops
  - Examples
    - Print "hello" 10 times
    - Find all the prime numbers up to an integer n
    - Print each odd number between 5 and 127

- **Indefinite loop:** Number of repeats is *not known in advance*
  - Examples
    - Prompt the user until they type a non-negative number
    - Print random numbers until a prime number is printed
    - Repeat until the user types "q" to quit

while Loop

- **while loop: Repeatedly executes its body *while* a logical test is true**

```
while (test) {
    statement(s);
}
```



- **Example**

```
int num = 1;                    // initialization
while (num <= 200) {            // test
    System.out.print(num + " ");
    num *= 2;                   // update
}
// output: 1 2 4 8 16 32 64 128
```

Infinite loop with `while`

- **The test is checked every time!**

```
while (true) {
    System.out.println("Stop!!!");
}
```

- Press **Ctrl + C** to exit out of programs that don't stop (on their own)

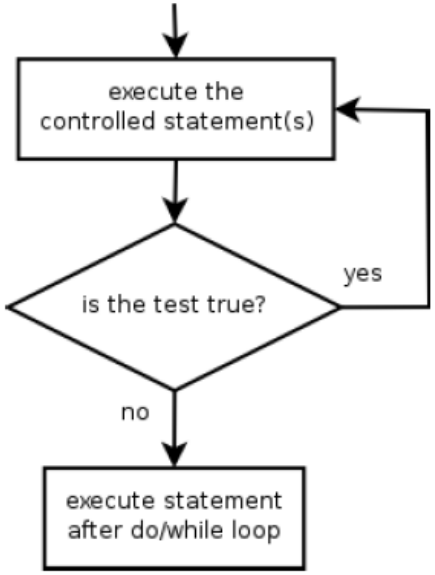
- Commonly found when ***updating procedure*** is not found

```
int num = 1;                    // initialization
while (num <= 200) {            // test
    System.out.print(num + " ");
    // num *= 2;                // no update
}
// output: 1 1 1 1 1 1 1 1 1 ...
```

do-while loop

- **Execute it body once, and execute it again *while* the test is true**
  - Performs its test at the ***end*** of each repetition
  - Guarantees that the loops body will run ***at least once***
  - Must end with a semicolon after `while`

```
do {
    statement(s);
} while (test);
```



- **Example**

```
Scanner sc = new Scanner(System.in);
int x;
do {
    System.out.print("Type in a number less than 10: ");
    x = sc.nextInt();
} while(x >= 10);
System.out.println("OK");
```

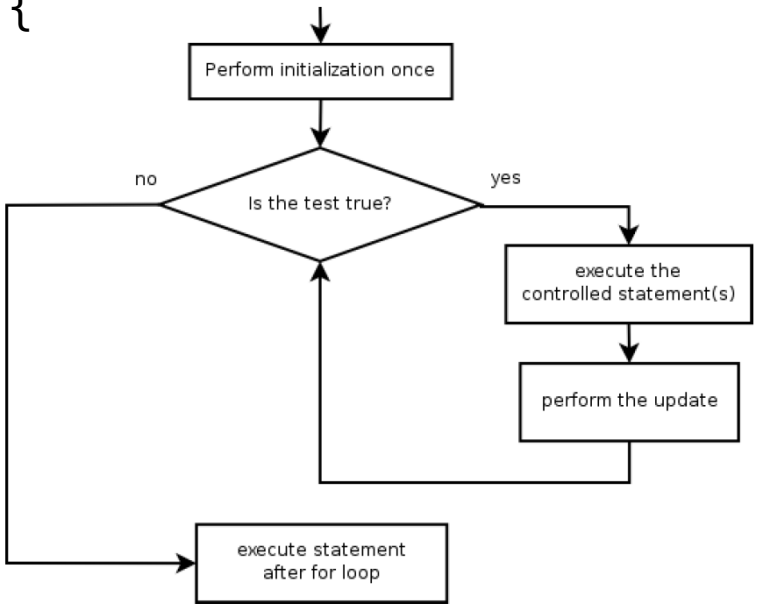
Exercise

- #2741 N 찍기

- #10950 A + B – 3

for loop

```
for (initialization; test; update) {
    statement(s);
}
```



- **Perform initialization once**
- **Repeat:**
  - Check if the **test** is true. If false, stop
  - Execute the **statements**
  - Perform the **update**

for loop - Initialization

```
for(int i = 1; i <= 6; ++i) {
    System.out.println("For Example");
}
```

- **Tells Java what variable to use in the loop**
  - Performed **once** as the loop begins
  - The variable is called a *loop counter*
    - Can use other variable names
    - Can start at any value
    - Can initialize many variables at once

for loop - Test

```
for(int i = 1; i <= 6; ++i) {
    System.out.println("For Example");
}
```

- **Tests the expression**
  - Must be a boolean expression (evaluates to either true or false)
    - Can use complex boolean expressions
  - If true, execute the block
  - If false, stop

for loop - Update

```
for(int i = 1; i <= 6; ++i) {
    System.out.println("For Example");
}
```

- **Modify the loop counter**
  - Pre/Post increment/decrement operator is used often
  - Can modify the loop counter to any value

```
for (int i = 1, j = 1; i + j <= 13; ++i, j += 2) {
    System.out.println(i + " " + j);
}
```

Infinite loop with for

- **These are possible, and will not stop**

```
for(;;) {
    System.out.println("Hello, Java");
}

for(; true; ) {
    System.out.println("Hello, Java");
}
```

Exercise

- #2739 구구단
- #2742 기찍 N
- #10871 X 보다 작은 수

Nested for loops

```
for(int i = 1; i <= 5; ++i) {
    for(int j = 1; j <= 10; ++j)
        System.out.print("*");
    System.out.print('\n');
}
```

- **Output**

```
**********
**********
**********
**********
**********
```

- **The inner loop executes 10 times, outer loop executes 5 times**

Exercise

- #2438 별 찍기 – 1

- #2439 별 찍기 – 2

break Statement

- Used to *break out* of `for`, `while`, `do-while` loops

```
for (int i = 1; i <= 10; ++i) {
    System.out.println(i);
    if (i == 3)
        break;
}
System.out.println("Done");
```

- Breaks out of loop and executes the next statement

break Statement

- In nested loops, break only breaks out of a single loop

```
for (int i = 1; i <= 3; ++i) {
    for (int j = 1; j <= 10; ++j) {
        if (j == 2)
            break;
        System.out.println("j: " + j);
    }
    System.out.println("i: " + i);
}
System.out.println("Done");
```

- Breaks out of loop and executes the next statement

Exercise

- **#10952 A + B – 5**

`continue` Statement

- **Used to skip the rest of the statement and execute the next loop**

- **Example: Print odd integers from 1 to 10**

```
for (int i = 1; i <= 10; ++i) {
    if (i % 2 == 0)
        continue;
    System.out.println(i);
}
```

  - If `i` is even, print statement is skipped

for/while Conversion

- **for loops and while loops are interchangeable!**

```
for (initialization; test; update) {
    statement(s);
}
```

```
initialization;
while (test) {
    statement(s);
    update;
}
```

Repeated Code Example

- **What's bad about this code?**

```
public class MethodExample1 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();
        for(int i = 0; i < 30; ++i)
            System.out.print("-");
        System.out.println();
        while(t-- > 0) {
            String s = sc.next();
            System.out.println(s);
            for(int i = 0; i < 30; ++i)
                System.out.print("-");
            System.out.println();
        }
    }
}
```

`static` Methods

- **static method: A named group of statements**
  - Denotes the *structure* of a program
  - Eliminates *redundancy* by code reuse
  - Procedural decomposition
    - Dividing a problem into methods
  - Writing a static method is like adding a new command

- **Steps**
  1. Design the commands
  2. Declare (write down) the methods
  3. Call (run) the methods

`static` Methods

- **Declare:** Give your method a name so it can be executed
- **Syntax**

```
public static void name() {
    statement(s);
}
```

- **Call:** Execute the method's code by calling
  - You can call as many times as you want

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

Control Flow

- **When a method is called, the program's execution**
  - "jumps" into that method
  - executes the methods statements
  - "jumps" back to the point where the method was called

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

public static void name() {
    statement(s);
}

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

Repeated Code Example - Better

- **What's bad about this code?**

```
public class MethodExample2 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();
        line();
        while(t-- > 0) {
            String s = sc.next();
            System.out.println(s);
            line();
        }
    }

    public static void line() {
        for(int i = 0; i < 30; ++i)
            System.out.print("-");
        System.out.println();
    }
}
```

Scope of Variables

- Suppose we want to change the length of the line **each time**
  - Receive an input from the user
- ***This doesn't work... Why?***

```
public class MethodExample2 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();
        line();
        while(t-- > 0) {
            String s = sc.next();
            System.out.println(s);
            line();
        }
    }

    public static void line() {
        t = sc.nextInt();
        for(int i = 0; i < 30; ++i)
            System.out.print("-");
        System.out.println();
    }
}
```

Scope of Variables

- **scope: The part of a program where a variable exists**
  - Usually ***from its declaration to the end of the { } braces***
    - A variable declared in a `for` loop exists only in that loop
    - A variable declared in a method exists only in that method

- **Example**

```
public static void example() {
    int x = 3;
    for (int i = 1; i <= 10; ++i) {
        System.out.println(x);
    }
    // i no longer exists here
}
// x no longer exists here
```

Scope of Variables

- **Variables without overlapping scope can have same names**

```
for (int i = 1; i <= 100; ++i) {
    System.out.println('/');
}
for (int i = 1; i <= 100; ++i) {
    System.out.println('\\');
}
int i = 5;                  // OK: outside of loop
```

- **A variable can't be declared twice or used out of its scope**

```
for (int i = 1; i <= 100; ++i) {
    int i = 2;                  // Error: duplicate local variable
    System.out.println('/');
}
i = 4;              // Error: i cannot be resolved to a variable
```

Scope of Variables

- **You can use variables that can be accessed anywhere in the class**
  - Its **scope** is the whole class

```
public class ScopeExample2 {
    public static int x = 5;

    public static void main(String[] args) {
        System.out.println(x);          // OK
    }

    public static void foo() {
        System.out.println(x + 1);      // OK
    }
}
```

- **Syntax**
  - `public static type name = value;`
- Generally not recommended – Best to keep scopes *small as possible*
  - Use ***class constants*** (or just don't use them an find another way)

Class Constants

- ***constant*: A variable where its value can be set only at declaration**
  - Cannot be reassigned
  - Uses **`final`** keyword
- ***class constant*: A constant visible to the whole class**
  - Name is usually in ALL_UPPER_CASE
  - Use it only when necessary!

```
public class ScopeExample3 {
    public static final int SIZE = 5;
    public static final double PI = 3.1415;
    public static final String MY_NAME = "Name";

    public static void main(String[] args) {
        final int x = 5;
        x = 3;   // Error: cannot be reassigned
    }
}
```

Fixed Code

- **Declared Scanner as a class constant**

```
public class MethodExample3 {
    public static final Scanner sc = new Scanner(System.in);

    public static void main(String[] args) {
        int t = sc.nextInt();
        line();
        while (t-- > 0) {
            String s = sc.next();
            System.out.println(s);
            line();
        }
    }

    public static void line() {
        int len = sc.nextInt();
        for (int i = 0; i < len; ++i)
            System.out.print("-");
        System.out.println();
    }
}
```

Redundant Code

- **Too much input ...**
  - Fix the length of line to be 10 using a class constant
  - Change Scanner to local variable
- **What if we..**
  - Wanted to print a line of stars (*) instead of dashes (-) ?
  - ***Need to declare a method for different characters***

```
public class MethodExample4 {
    public static final int LEN = 10;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();
        star();
        while (t-- > 0) {
            String s = sc.next();
            System.out.println(s);
            tilde();
        }
        line();
    }
    public static void line() {
        for (int i = 0; i < LEN; ++i)
            System.out.print("-");
        System.out.println();
    }
    public static void star() {
        for (int i = 0; i < LEN; ++i)
            System.out.print("*");
        System.out.println();
    }
    public static void tilde() {
        for (int i = 0; i < LEN; ++i)
            System.out.print("~");
        System.out.println();
    }
}
```

Parameterization

- ***parameter*: A value passed to a method by its caller**
  - Instead of declaring method for different character, write a method to print any string repeatedly
    - When declaring the method, state that it requires a string parameter
    - When calling the method, specify the string to print

- **Syntax**
  - Declaration

```
public static void name(type param1) {
    statement(s);
}
```

  - Passing a parameter
    - The value of expression must match the type in declaration

```
name(expression);
```

Redundant Code - Fixed

- **Declare**
  - `line(String str)`
  - The method will print the passed string 10 times

```
public class MethodExample5 {
    public static final int LEN = 10;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();
        line("*");
        while (t-- > 0) {
            String s = sc.next();
            System.out.println(s);
            line("~");
        }
        line("-");
    }

    public static void line(String str) {
        for (int i = 0; i < LEN; ++i)
            System.out.print(str);
        System.out.println();
    }
}
```

Parameterization – Multiple Parameters

- **Can pass multiple parameters to a method**

```
public static void line(String str, int len) {
    for (int i = 0; i < len; ++i)
        System.out.print(str);
    System.out.println();
}

line("???", 5);  // prints ???????????????
```

- **Syntax**
  - Declaration

```
public static void name(type param1, ..., type paramn) {
    statement(s);
}
```

  - Passing parameters

```
name(expr1, ..., exprn);
```

Parameterization – Common Errors

- **If a method accepts a parameter, it is illegal to call it without passing any value for that parameter**

```
line();    // Error: parameter required
```

- **The value passed to a method must be of the correct type**

```
line("a", 3.2);       // Error: must be of type int
```

Value Semantics

- **When the method is called:**
  - The value is stored into the parameter variable
  - The method's code executes, using that value (inside variable)

- **value semantics: When primitive values are passed as parameters, *their values are copied***
  - Modifying the parameter **will not affect the variable passed in**

```
public static void main(String[] args) {
    int x = 23;
    strange(x);
    System.out.println(x);   // 23
}

public static void strange(int x) {
    x = x + 1;
    System.out.println(x);   // 24
}
```

Value Semantics

- **Example**

```
public static void main(String[] args) {
    int x = 10, y = 5;
    swap(x, y);
    System.out.println(x + ", " + y);
    // 10 5 (not swapped)
}

public static void swap(int x, int y) {
    int tmp = y;
    y = x;
    x = tmp;
    System.out.println(x + ", " + y);    // 5 10
}
```

Exercise

- main 이외의 static method 를 사용할 것

- #2440 별 찍기 – 3

- #2442 별 찍기 – 5

Return

- **return: To send out a value as the result of a method**
  - Parameters send info ***in*** from the caller to the method
  - Return values send information ***out*** of a method to its caller
  - The method call will be evaluated to its return value

- **Syntax**
  - Specify the return type in declaration
    - `void` does not return anything
  - Method must return a value according to the type in declaration

```
public static type name(parameters) {
    statement(s);
    return expression;
}
```

Return Example

- **Absolute value function**

```
public static double abs(double x) {
    if (x >= 0)
        return x;
    else
        return -x;
}
```

  - `abs(1.2)` will be evaluated to 1.2, with type `double`
  - Can store the return value of the method by
    - `double y = abs(x);`
    - Now, this y can be used in other expressions

Exercise

- main 이외의 static method 를 사용할 것
- #4673 셀프 넘버
- #1065 한수

Exception

- ***exception*: An object representing a runtime error**
  - We say that a program with an error throws an exception
  - It is also possible to catch (handle) an exception

- ***checked exception*: An error that must be handled by our program (otherwise it will not compile)**
  - Must specify how the program will behave if the exception occurs
  - Unchecked exception do not have to be handled
    - But the code needs to be fixed

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at examples.ExceptionExample1.main(ExceptionExample1.java:7)
```

throws

- **`throws`: Keyword on a method's header that states that *the method may generate an exception (and will not handle it)***
  - "I hereby announce that this method might throw an exception, and I accept the consequences if this happens"

- **Example**

```
public static void main(String[] args) throws ArithmeticException {
    ...
}
```

`try-catch` Statement

- **Syntax**

```
try {
    statement(s);
} catch (exception e) {
    statement(s);
}
```

- **While executing the statements in `try`,**
  - If an exception ***specified*** in the `catch` statement occurs,
  - The exception is *caught* and `catch` block is executed
  - If exception doesn't occur, `catch` block is ignored
  - You can catch multiple exceptions by adding more `catch` blocks

try-catch Statement

- **Example**

```
public static void main(String[] args) {
     try {
          int k = area(3, -5);
     } catch (IllegalArgumentException e) {
          System.out.println(e.getMessage());
          e.printStackTrace();
     }
}

public static int area(int x, int y) {
     if(x < 0 || y < 0)
          throw new IllegalArgumentException("Length cannot be negative");
     return x * y;
}
```

- **You can also throw an exception**
  - And include a custom error message

Exercise

- **Write a program that takes two integers x, y and prints $x \ / \ y$**
  - Conditions:
    - Use a static method that takes two integers x, y and returns x/y
    - The method should detect and throw an ArithmeticException if division by 0 occurs
    - The exception should contain the message
      - "Division by 0"
    - The main method should use try-catch to handle the ArithmeticException
      - In the catch block, print the message of exception

System.out.printf

- **Used to format when printing text**
  - Does not produce new line at the end
- `System.out.printf("format string", parameters);`
  - There can be many parameters
- **A format string can contain placeholders to insert parameters**
  - %d integer
  - %f real numbers (double)
  - %.Df real numbers, with D digits precision
  - %s string
- **Example**
  - `System.out.printf("My name is %s %s", firstName, lastName);`
  - `System.out.printf("The value of %s is %d", "x", x);`
  - `System.out.printf("%.3f", 3.141592); // 3.142`

Exercises

- Use `printf`

- #11021 A + B – 7

- #3053 택시 기하학

Homework

- **Textbook Chapter 1 Multiple-Choice questions**