COMP2026

Problem Solving Using Object Oriented Programming

Methods

Outline

- Basics of Methods
- Syntax of Methods
- Return statement
- Arguments and Parameters
- Methods and Arrays
- Method Overloading
- Problem Solving with Methods

Basics of Methods

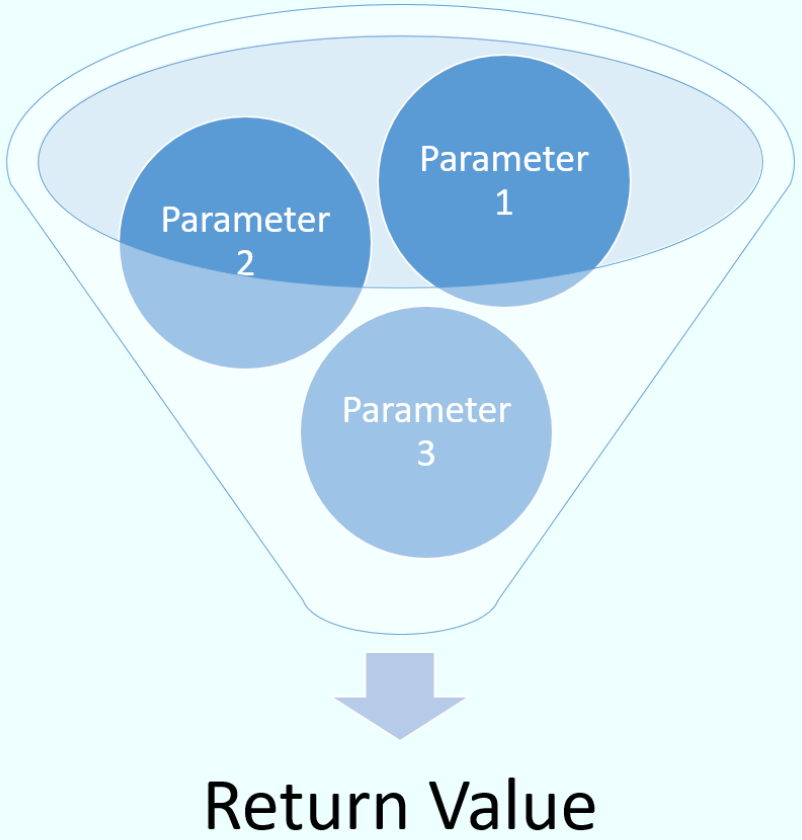
- The term **functions** is called as **methods** in Java.
- A **method** is designed and used to represent some **repeating logic** in a programming code.
- Mathematically, a function can be defined as

$$f(x) = x^3 + 1$$
$$g(x, y) = xy + \sin y$$

- So, for instance, we can use the expression $g(3, \pi)$ to represent the equation $3\pi + \sin \pi$.

Basic of Methods

- A math function has a number of inputs and one output
- A Java method has a number of inputs called parameters and one output return value



Basics of Methods

- A method is a block of statements that performs a specific task, and we assign a name to it.

```
class FunctionExample {
    int f(int x) {  //f is a method
        return x * x * x + 1;
    }
    void runOnce() { //runOnce is also a method
        int result1 = f(5);
        System.out.println("f(5) = " + result1);  //print f(5) = 126
        int result2 = f(10);
        System.out.println("f(10) = " + result2); //print f(10) = 1001
        System.out.println("f(20) = " + f(20)); //print f(20) = 8001
    }

    public static void main(String[] argv) { //main is a special method
        new FunctionExample().runOnce();
    }
}
```

Basics of Methods

- When a method is called, the executation *branches to the method* and executes the block of statements in that method.
- When the method finishes, it returns to the position of where it is called.

```
class FunctionExample {
    int f(int x) {  //f is a method
        return x * x * x + 1;
    }
    void runOnce() {//runOnce is also a method
        int result1 = f(5);
        System.out.println("f(5) = " + result1);  //print f(5) = 126
        int result2 = f(10);
        System.out.println("f(10) = " + result2); //print f(10) = 1001
        System.out.println("f(20) = " + f(20)); //print f(20) = 8001
    }

    public static void main(String[] argv) { //main is a special method
        new FunctionExample().runOnce();
    }
}
```

Syntax of Methods

- A method has the following parts

```
returnType functionName (parameterType parameterName){
    ...method body...
    return returnValue;
}
```

```
char gradeMethod(int x) {
    char grade = ' ';
    if (x > 70)
        grade = 'A';
    else if (x > 50)
        grade = 'B';
    else if (x > 35)
        grade = 'C';
    else
        grade = 'F';
    return grade;
}
```

- **return type:** `char`
- **function name:** `gradeMethod`
- **parameter type:** `int`
- **parameter name:** `x`
- **return value:** `grade`

Return value

- The type of the **return value** must match with the **return type**.
- i.e. `char gradeMethod(int x)` would expect returning a char.

```
char gradeMethod(int x) {
    return 'A'; //OK
}
String greeting(String name) {
    System.out.println("Greeting!");
    return "Hello, " + name; //OK
}

int f(int x) {
    return "COMP2026"; //error! Return type mismatch
}
```

Return value

- Codes after the return statement will not be executed.

```
char gradeMethod(int x) {
  char grade = ' ';
  if (x > 70)
      grade = 'A';
  else if (x > 50)
      grade = 'B';
  else if (x > 35)
      grade = 'C';
  else
      grade = 'F';
  return grade;
  return 'G'; //unreachable statement
  System.out.println("Should not see this"); //unreachable statement
}
```

Return value

- A method may have multiple `return` on different branches.

```
char gradeMethod(int x) {
    if (x > 70)
        return 'A';
    else if (x > 50)
        return 'B';
    else if (x > 35)
        return 'C';
    else
        return 'F';
}
```

```
char gradeMethod(int x) {
    if (x > 70)
        return 'A';
    if (x > 50)
        return 'B';
    if (x > 35)
        return 'C';
    return 'F';
}
```

🧑‍🏫💬 They are the same. Why?

Return value

- Except for `void` methods, all branches in a method must be terminated by a return statement.

```
String luckDraw(int ticketNumber) {
    if (tickerNumber == 2026)
        return "1st Prize";
    if (ticketNumber % 5 == 0) {
        if (ticketNumber % 3 != 0)
            return "4th Prize";
        else if (ticketNumber % 2 == 0)
                return "3rd Prize";
        return "2nd Prize";
    }

}
```

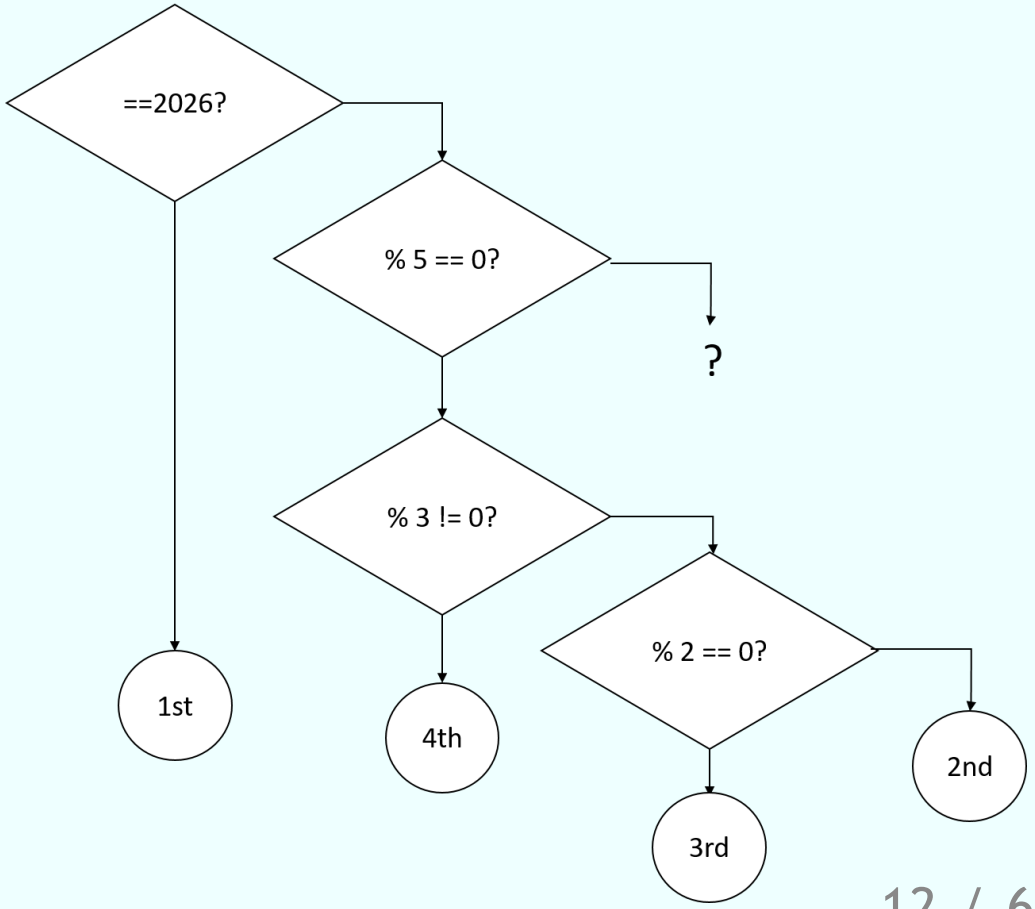
Something wrong here...

Return value

- Except for `void` methods, all branches in a method must be terminated by a return statement.

```
String luckDraw(int ticketNumber) {
    if (tickerNumber == 2026)
        return "1st Prize";
    if (ticketNumber % 5 == 0) {
        if (ticketNumber % 3 != 0)
            return "4th Prize";
        else if (ticketNumber % 2 == 0)
                return "3rd Prize";
        return "2nd Prize";
    }

    return "No prize";
}
```



Void Method

- A void method does not return a value.
- It only performs certain tasks or routines.
- A void method has the return type `void`.

```
void greeting(String name) {
    System.out.println("Hello, " + name + "!");
}
```

```
void grade(int score) {
    if (score > 70)
        System.out.println("A!");
    else if (score > 40)
        System.out.println("Pass!");
    else
        System.out.println("Fail!");
}
```

Void Method

- Return statement in a void method is **not mandatory**.
- Return statement only exit the function. It does *not carry a return value*.
- Return statement in a void function is simply `return;` (no value after return).

```java
void grade(int score) {
    if (score > 70)
        System.out.println("A!");
    else if (score > 40)
        System.out.println("Pass!");
    else
        System.out.println("Fail!");
}
```

```java
void grade(int score) {
    if (score > 70) {
        System.out.println("A!");
        return;
    }
    if (score > 40) {
        System.out.println("Pass!");
        return;
    }
    System.out.println("Fail!");
}
```

Void method

- Return statement in a void method is **not mandatory**.
- Return statement only exit the function. It does *not carry a return value*.
- Return statement in a void function is simply `return;` (no value after return).

```
void grade(int score) {
    if (score > 70) {
        System.out.println("A!");
        return 'A'; //error!
    }
    if (score > 40) {
        return System.out.println("Pass!"); //error!
    }
    System.out.println("Fail!");
}
```

Methods without a parameter

- It is possible that a method does not have any parameter.

```
String enterPassword() {
    Scanner scanner = new Scanner(System.in);
    String password;
    do {
        System.out.println("Please enter your new password");
        password = scanner.next();
        System.out.println("Please enter your password again");
    } while (!password.equals(scanner.next()));
    return password;
}

void runOnce() {
    System.out.println("Your new password is " + enterPassword());
}
```

- Add an empty bracket `()` after your parameterless method when you define it or call it.

Methods with more than one parameters

- It is possible that a method has more than one parameters.
- Parameters are separated by `,`
- Each parameter requires its own type.

```
String boyFriendSelector(int month, int day) {
    if ((month == 11 && day >= 22) || (month == 12 && day <=21))
        return "Sagittarius, hmm, worth a try";
    if ((month == 4 && day >= 20) || (month == 5 && day <=20))
        return "My-ex was a Taurus, never consider!";
    return "Not sure";
}

void runOnce() {
    System.out.println(boyFriendSelector(5,4)); //4th May.
}
```

Methods with more than one parameters

- It is possible that a method has more than one parameters.
- Parameters are separated by `,`
- Each parameter requires its own type.

```
String bfSelector(int m, d) {
    //error!
}
```

❌ unlike declaring variables, each parameter must has its own type

```
String bfSelector(int m, int d) {
    //correct
}
```

✅ add the parameter type for each parameter

Calling a method - argument vs parameter

```
void runOnce() {
    sayHello("Kevin", "Wang");
}
void sayHello(String fName, String lName) {
    System.out.println("Hello " + fName + " " + lName + "!");
}
```

- Caller - the line calling the method (`runOnce()`)
- Callee - the method being called (`sayHello`)
- Arguments - values sent from the caller to the callee ("Kevin", "Wang")
- Parameters - variables that receive values from the caller by the callee (`fName`, `lName`)
- A method that accepts **arguments** has **parameters** to receive the arguments

Calling with correct parameter

- When calling a method, you need to supply the **same number** of argument in **correct order** with **correct type** that matches the method's parameters

```
double log(double x, int base) {...}
void runOnce() {
    log(4.33, 5); //OK
    log(4.33); //error! insufficient argument
    log(4.33, 5, 10); //error! too many argument
    log(5, 4.33); //error! arguments are not in correct order
    log(4.33, "Hello"); //error! incorrect type
}
```

- Note that in the method call, we do not include the data type of the argument in the parentheses

```
log(double 4.33, int 5); //incorrect
```

Scope of a parameter

- Recall that a scope of a variable is confined to where it is declared.
- Parameter is not visible outside a function

```
void grade(int score) {
    String result = "";
    if (score > 70)
        result = "A";
    else if (score > 40)
        result = "Pass!";
    else
        result = "Fail!";
    System.out.println(result);
}

void runOnce() {
    grade(50);
    System.out.println(result); //error! local variable of method grade is invisible
    System.out.println(score); //error! parameter of method grade is invisible
}
```

Parameters being modified

- What will happen if the parameter is modified?

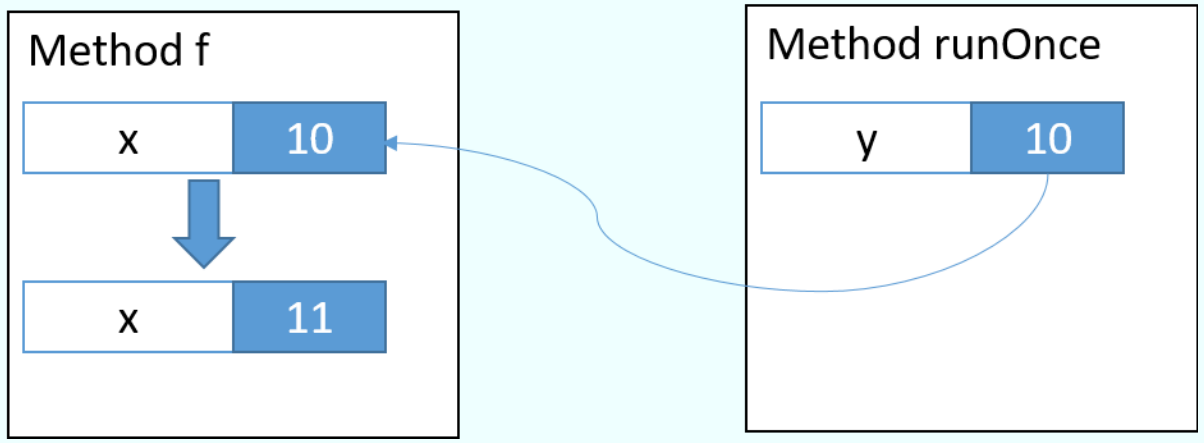
```
void f(int x) {
    System.out.println("x = " + x);
    x++;
    System.out.println("x = " + x);
}

void runOnce() {
    int y = 10;
    System.out.println("y = " + y);
    f(y);
    System.out.println("y = " + y);
}
```

```
y = 10
x = 10
x = 11
y = 10
```

Parameters being modified

- Each method has its own memory spaces.
- What happen in a method calling is the value of the argument in the caller is **copied** to the parameter.
- When the value in the parameter is updated, it does not affect the memory inside the caller (runOnce)



This makes more sense

```
void f(int x) {
      x++;
}
void runOnce() {
     f(10); //the constant literal can't be changed!
}
```

Parameters being modified

- Even when the parameter and the argument has the same name, the argument will not be affected!

```
void f(int x) {
    System.out.println("(f) x = " + x);
    x++;
    System.out.println("(f) x = " + x);
}
void runOnce() {
    int x = 10;
    System.out.println("(runOnce) x = " + x);
    f(x);
    System.out.println("(runOnce) x = " + x);
}
```

```
(runOnce) x = 10
(f) x = 10
(f) x = 11
(runOnce) x = 10
```

Parameters being modified

```java
void runOnce() {
    String name = "Kevin";
    changeName(name);
    System.out.println(name);
}

void changeName(String name) {
    System.out.println("Name inside changeName: " + name);
    name = "Calvin";
    System.out.println("Change my name inside method: " + name);
}
```

```
Name inside changeName: Kevin
Change my name inside method: Calvin
Kevin
```

Local variable of a method

```
void goldfish() {
    int x = 10;
    System.out.println(x);
    x++
    System.out.println(x);
}
void runOnce() {
    goldfish();
    goldfish();
}
```

```
10
11
10
11
```

- Local variables defined in a method or the parameter will not be persistent.
- A method's memory will be wiped once the method is returned.

More on Arguments

- Any expression with a value that could be assigned to a variable of the parameter's data type may be used as the argument in the method call

```
void printGrade(int score) {...} //a method that print grades, not very important
void runOnce() {
    printGrade(99); //OK
    printGrade(198 / 2 - 49); //OK
    int score = 33; //don't worry about name clash
    printGrade(score);  //OK
}
```

- When you pass an argument to a method, you must ensure that the argument's data type is **compatible** with the data type of the parameter
- Java will automatically perform widening conversions. This means that if the argument is of a lower-ranking data type than the parameter, the argument will automatically be converted to the parameter's data type (see **casting**)

More on Arguments

- Java does not automatically perform narrowing conversions, conversions to lower-ranking data types
- If you try to pass an argument of a higher-ranking data type into a parameter variable, a compiler error occurs
- You may use the cast operator to manually specify a conversion to a lower-ranking data type

```
private double multiply(int a, double b) {
    return a * b;
}
multiply(16 + 7, 31.3); //OK
multiply(16 + 7, 31); //OK widening from 31 (int) to double
multiply(16.5, 31.3); //error! Narrowing cause lost of precision
multiply((int) 16.5, 31.3); //OK, manually cast
```

Methods and Arrays

- Must be careful when passing an array to a method
- Suppose an array is passed as an argument to a method:

```
int[] anArray = new int[10];;
someMethod(anArray);
```

1. In the method, elements of the array can be modified.
2. When the element of the array is modified, the argument `anArray` is also modified.
3. The method can also return a new array:

```
int[] theReturnedArray = methodThatReturnsArray();
```

Methods and Arrays - as parameter

```
void doubleItUp(int[] array) {
    for (int i = 0; i < array.length; i++) {
        array[i] = array[i] * 2;
    }
}
void runOnce() {
    int [] myArray = { 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 };
    doubleItUp(myArray);
    for (int i : myArray)
        System.out.printf("%d ", i);
}
```

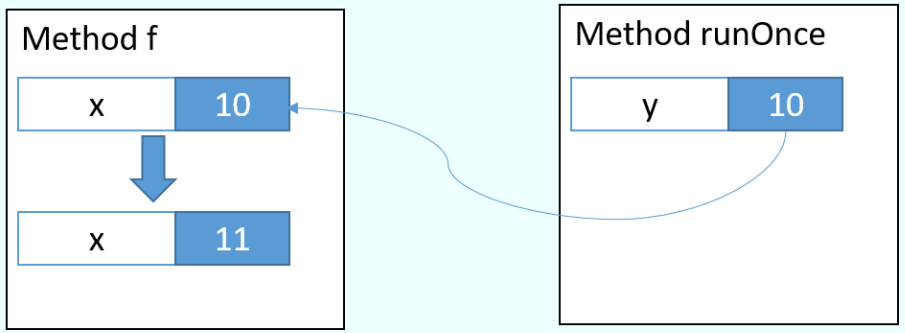
```
2 8 18 32 50 72 98 128 162 200
```

- You can modify the content of an array you pass as an argument to a method
- Note that the method returns nothing

Methods and Arrays - Recalls..

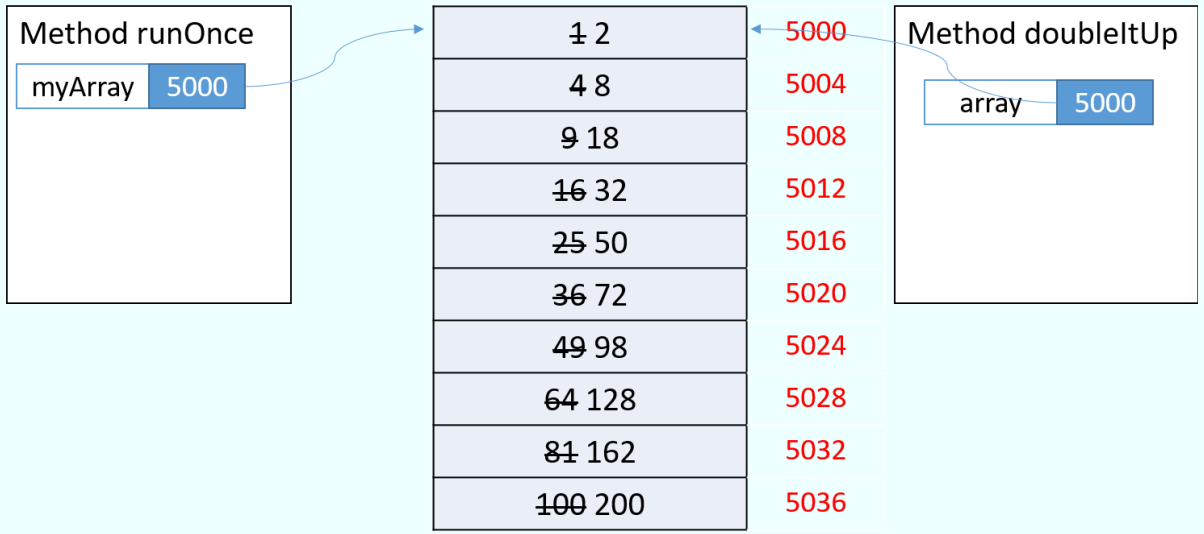
```
void f(int x) {
    System.out.println("x = " + x);
    x++;
    System.out.println("x = " + x);
}
void runOnce() {
    int y = 10;
    f(y);
    System.out.println("y = " + y);
}
```

- Each method has its own memory spaces.
- What happen in a method calling is the value of the argument in the caller is **copied** to the parameter.
- When the value in the parameter is updated, it does not affect the memory inside the caller (runOnce)



Methods and Arrays - as parameter

```
void doubleItUp(int[] array) {
  for (int i = 0; i < array.length; i++) {
    array[i] = array[i] * 2;
  }
}
void runOnce() {
  int [] myArray = { 1, 4, 9, 16, 25, 36,
    49, 64, 81, 100 };
  doubleItUp(myArray);
  for (int i : myArray)
    System.out.printf("%d ", i);
}
```



- The array is declared and allocated in `runOnce`
- `myArray` holds the *reference* of the array
- The reference is copied into the method `doubleItUp`
- `array` also holds the **same reference**

Methods and Arrays - as return value

```
void runOnce() {
    int[] callerArray = squareNum();
    ...
}
int[] squareNum() {
    int[] localArray = new int[5];
     for (int i = 0; i < localArray.length; i++) {
       localArray[i] = i * i;
    }
    return localArray;
}
```

```
localArray:
0, 1, 4, 9, 16
callerArray:
0, 1, 4, 9, 16
```

- The method is responsible for creating the new array and return the new array to its caller

Methods and Arrays - as return values

```
void runOnce() {
    int[] callerArray = squareNum();
    ...
}
int[] squareNum() {
    int[] localArray = new int[5];
     for (int i = 0; i < localArray.length; i++) {
       localArray[i] = i * i;
    }
    return localArray;
}
```

0	5000
1	5004
4	5008
9	5012
16	5016

- The array is first declared and allocated as a local variable `localArray` inside squareNum
- The reference of localArray is **copied** to caller's array variable `callerArray`.
- Both `localArray` and `callerArray` points to the same array.

Methods and Arrays - as return values

```
void runOnce() {
    int[] callerArray = squareNum();
    int[] callerArray2 = squareNum();
}
int[] squareNum() {
    int[] localArray = new int[5];
     for (int i = 0; i < localArray.length; i++) {
       localArray[i] = i * i;
    }
    return localArray;
}
```

`callerArray`

0	5000
1	5004
4	5008
9	5012
16	5016

`callerArray2`

0	7200
1	7204
4	7208
9	7212
16	7216

- The variable `localArray`, which stores the reference of the array, will be erased when the method returns. The array will be persistent.
- When the second time `squareNum()` is called, another array is declared and allocated.
- `localArray` will store the second array
- `callerArray2` stores the references of the second array

Methods and Arrays - Mixed together

- A method can process an array and return a completely new array

```
void runOnce() {
    int[] array1 = { 1, 2, 3 };
    int[] array2 = enlargeArray(array1);
}
int [] enlargeArray(int[] origArray) {
    int[] newArray = new int[origArray.length*2];

    for (int i = 0; i < origArray.length; i++) {
       newArray[i] = origArray[i];
       newArray[i+origArray.length] = origArray[i] * 2;
    }
    return newArray;
}
```

```
array1:
1, 2, 3
array2:
1, 2, 3, 2, 4, 6
newArray:
1, 2, 3, 2, 4, 6
```

- The variable `newArray`, which holds the reference, will be erased after the function. The array will be persistent.

Methods and Arrays - Exercise #1

What is the expected value of `arg` and `x`?

```
int f(int param) {
    param++;
    return param + 1;
}
void runOnce() {
    int arg = 10;
    int x = f(f(arg));
}
```

Methods and Arrays - Exercise #2

What is the expected value of `arr` and `x`?

```
int f(int[] param) {
    param[0]++;
    return param[0] + 1;
}
void runOnce() {
    int[] arr = {10};
    int x = f(arr);
}
```

Methods and Arrays - Exercise #3

What is the expected value of `arr` and `x`?

```
int f(int[] param) {
    param[0]++;
    return param[0] - 1;
}
void runOnce() {
    int[] arr = {10};
    arr[0] = f(arr);
    int x = f(arr);
}
```

Methods and Arrays - Exercise #4

👨‍🏫💬 What is the expected value of `array1`, `origArray`, `newArray`?

```
void runOnce() {
    int[] array1 = { 1, 2, 3 };
    enlargeArray(array1);
}
void enlargeArray(int[] origArray) {
    int[] newArray = new int[origArray.length * 2];

    for (int i = 0; i < origArray.length; i++) {
        newArray[i] = origArray[i];
        newArray[i+origArray.length] = origArray[i] * 2;
    }

    origArray = newArray;
}
```

Methods and Arrays - Exercise #5

👨‍🏫💬 This program crashes. Why?

```
void runOnce() {
    int[] array1 = { 1, 2, 3 };
    enlargeArray(array1);
}

void enlargeArray(int[] origArray) {
    origArray = new int[origArray.length*2];

    for (int i = 0; i < origArray.length; i++) {
        origArray[i] = origArray[i];
        origArray[i+origArray.length] = origArray[i] * 2;
    }

}
```

Return array vs array parameter

- Both returning an array and using array parameter allows you to modify the content of an array.

```
int[] method(int[] input) {
    ...
    return newArray;
}
```

```
void method(int[] input) {
    ...
    input[0] = 10;
}
```

- Which to use?
- When you don't want to modify the original array (i.e. making a new copy is needed), use return
- When you are going to modify the size of the original array (esp, enlarging), use return
- When you are going to change the value of the original array, use it as a parameter and change it directly.

Method overloading

Method overloading

- You can define two methods with the same name but with different parameter list.
- This is called **method overloading**.

```
void printResult(int mark) {
    System.out.println("You have got " + mark + " marks!");
}
void printResult(char grade) {
    System.out.println("You have got a " + grade + "!");
}
double printResult(int mark, int max) {
    System.out.printf("You have got %d/%d !\n", mark, max);
    return mark * 1.0 / max;
}
```

- Purpose of overloading is to provide different response against different types of input for similar functions.

Method overloading

- Overloading is to be resolved when the method is called

```
void printResult(int mark) {
    System.out.println("You have got " + mark + " marks!");
}
void printResult(char grade) {
    System.out.println("You have got a " + grade + "!");
}
void runOnce() {
    printResult('A');
    printResult(33);
    printResult((int)'A');
}
```

```
You have got a A!
You have got 33 marks!
You have got 65 marks!
```

Method overloading

- Overloading can lead to ambiguity

```
void overload(int a, double b) { System.out.println("A");}
void overload(double a, int b) { System.out.println("B");}
//no problem when it is defined


void runOnce() {
    overload(5, 3.4); //OK, A
    overload(5.2, 3); //OK, B
    overload(4.3, 1.1); //error! can't compile
}
```

More about Method Overloading

- Overload methods are characterized by **method name** and **parameter lists.**
- Cannot define two methods with the same name, same parameter list, even if the return types are different or the name of parameter variables are different

```
void overload(int a, int b) {}
void overload(int a) {} //OK
void overload(int x, int y) {} //error!
int overload(int x, int b) {} //error!
```

- Do not confuse **method overloading** with **method overriding** (discussed in the topic Class/Object)

Problem Solving with Methods

By Kevin 2022/2023

An analogy in Math

- How do you think about the following expression?

$$7 \times 5 + 7 \times (28 \times 3) - 4 \times 7$$

- Clumsy. This is what I will do

$$7(5 + 28 \times 3 - 4)$$

- Factorize your expression looks tidy and reduce number of computations.

Use methods to extract factors

- Using method is like taking some common factors from the code!

```java
for (int i = 0; i < 5; i++)
    System.out.print('*');
System.out.println();
for (int row = 1; row < 4; row++) {
      System.out.print(' ');
      for (int i = 0; i < 5; i++)
           System.out.print(' ');
      System.out.print('*');
}
for (int i = 0; i < 5; i++)
    System.out.print('*');
System.out.println();
```

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

```java
void printLine() {
    for (int i = 0; i < 5; i++)
        System.out.print('*');
    System.out.println();
}
...

printLine();

for (int row = 1; row < 4; row++) {
      System.out.print(' ');
      for (int i = 0; i < 5; i++)
           System.out.print(' ');
      System.out.print('*');
}

printLine();
```

Parameters

```java
if (boyHeight > 150)
    System.out.println("Tall");
else
    System.out.println("OK");
if (girlHeight > 130)
    System.out.println("Tall");
else
    System.out.println("OK");
```

```java
void printHeight(int height, int limit) {
    if (height > limit)
        System.out.println("Tall");
    else
        System.out.println("OK");
}
printHeight(boyHeight, 150);
printHeight(girlHeight, 130);
```

- Adding parameters to make the method be more general
- Don't waste any opportunity to *factorize* your code.

Return value

- All math functions have its return values
- Return values in methods are very useful
- e.g. Write a function for picking a password

```
String pickPassword() {
    String password ="";
    boolean valid = true;
    do {
        System.out.println("Pick a password");
        ... //let user input and validate it
    } while (!valid);
    return password;
}
...
String userPassword = pickPassword();
```

- The method not just validates users' inputs but also **guarantees and returns** a valid password to the caller.

Example - Printing Diamond

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

Strategy

- Defining methods that print a number of spaces

Example - Printing Diamond

```
void printSpace(int n) {
    for (int i = 0; i < n; i++)
        System.out.print(' ');
}
void runOnce() {
    int size = 4; //the height = 2 * size - 1
    printSpace(size - 1);
    System.out.println("*"); //top *
    for (int i = 1; i < size; i++) {
        printSpace(size - i - 1);
        System.out.print("*");
        printSpace(2 * i - 1);
        System.out.println("*");
    }
    for (int i = size - 2; i >= 1; i--) {
        printSpace(size - i - 1);
        System.out.print("*");
        printSpace(2 * i - 1);
        System.out.println("*");
    }
    printSpace(size - 1);
    System.out.println("*"); //bottom *
}
```

```
   *     --
  * *     |
 *   *    | - size
*     *   --
 *   *
  * *
   *
```

Example - Print Filled Diamond

```
    *
   *+*
  *+++*
 *+++++*
  *+++*
   *+*
    *
```

A very simple way to do it is to overload the method `printSpace`

```
void printSpace(int n, char c) {
    for (int i = 0; i < n; i++)
        System.out.print(c); //print the filling instead or space
}
```

Print Filled Diamond

```
void runOnce() {
    int size = 4; //the height = 2 * size - 1
    printSpace(size - 1);
    System.out.println("*"); //top *
    for (int i = 1; i < size; i++) {
        printSpace(size - i - 1);
        System.out.print("*");
        printSpace(2 * i - 1, '+'); //overload
        System.out.println("*");
    }
    for (int i = size - 2; i >= 1; i--) {
        printSpace(size - i - 1);
        System.out.print("*");
        printSpace(2 * i - 1, '+');  //overload
        System.out.println("*");
    }
    printSpace(size - 1);
    System.out.println("*"); //bottom *
}
```

Gotcha Simulator

- Assume there are 10 characters in a toy set.
- You pay a token to draw a random character until you have collected the entire characters.
- Estimate how many token you will need to spend.
- Implement that with methods.

Ingredient

- Array
- Random number generator - `ThreadLocalRandom.current().nextInt(0, 10);`

Gotcha Simulator

```
#7 Gotcha
#3 Gotcha
#1 Gotcha
#3 Duplicate
#7 Duplicate
#7 Duplicate
#0 Gotcha
#7 Duplicate
#5 Gotcha
#4 Gotcha
#6 Gotcha
#8 Gotcha
#9 Gotcha
#2 Gotcha
token spent:14
```

Gotcha Simulator

```java
boolean[] characters = new boolean[10];
int token = 0;
while (!collectAll(characters)) {
    //spend a token and draw
    int number = draw();
    //check if duplicate
    if (characters[number] == true)
        System.out.println("Duplicate");
    else
        System.out.println("Gotcha");
    characters[number] = true;
    token++;
}
```

👩‍🏫💬 **Do together**. write two methods: `collectAll` and `draw`

Gotcha Simulator

- Assume we are upgrading the simulator to conduct the experiment for 100 times and plot the histogram of the token spent.
- Instead of looping the code, how about **refactor** it as a method?

```
 0-  4 [ 0] |
 5-  9 [ 0] |
10- 14 [ 2] |**
15- 19 [17] |*****************
20- 24 [12] |************
25- 29 [23] |***********************
30- 34 [14] |**************
35- 39 [11] |***********
40- 44 [ 5] |*****
45- 49 [ 6] |******
50- 54 [ 3] |***
55- 59 [ 2] |**
60- 64 [ 2] |**
65- 69 [ 2] |**
70- 74 [ 1] |*
75- 79 [ 0] |
80- 84 [ 0] |
85- 89 [ 0] |
90- 94 [ 0] |
95-999 [ 0] |
```

Gotcha Simulator

```java
int sim() {
    boolean[] characters = new boolean[10];
    int token = 0;
    while (!collectAll(characters)) {
        //no need to print gotcha message
        characters[draw()] = true;
        token++;
    }
    return token;
}
```

```java
void runOnce() {
  int bins[] = new int[20];
  //bin size = 5, from 0 to 100+
  for (int i = 0; i < 100; i++) {
    int token = sim();
    if (token >= 100)
        bins[19]++;
    else
        bins[token / 5]++;
  }

  //print histogram
  for (int i = 0; i < 20; i++) {
    System.out.printf("%2d-%3d [%2d] |", i*5,
        (i == 19 ? 999 : i*5 + 4), bins[i]);
    for (int j = 0; j < bins[i]; j++)
        System.out.print('*');
    System.out.println();
  }
}
```

Tic-Tac-Toe with Methods

- Design three methods to separate the task of check wins

```
boolean checkRow(char[][] array, char symbol) {
    for (int row = 0; row < 3; row++)
        if (array[row][0] == array[row][1] && array[row][0] == array[row][2] &&
            array[row][0] == symbol)
            return true;
    return false;
}
boolean checkCol(char[][] array, char symbol) {
    for (int col = 0; col < 3; col++)
        if (array[0][col] == array[1][col] && array[0][col] == array[2][col] &&
            array[0][col] == symbol)
            return true;
    return false;
}
boolean checkDiagonal(char[][] array, char symbol) {
    if (array[1][1] != symbol) return false;
    if (array[0][0] == array[1][1] && array[1][1] == array[2][2]) return true;
    if (array[0][2] == array[1][1] && array[1][1] == array[2][0]) return true;
    return false;
}
```

Tic-Tac-Toe with Methods

```
//check horizontal
for (int row = 0; row < 3; row++)
    if (cells[row][0] == cells[row][1] &&
        cells[row][0] == cells[row][2] &&
        cells[row][0] == symbol)
      win = true;
//check vertical
for (int col = 0; col < 3; col++)
    if (cells[0][col] == cells[1][col] &&
        cells[0][col] == cells[2][col] &&
        cells[0][col] == symbol)
      win = true;
//check diagonal
if (cells[0][0] == cells[1][1] &&
   cells[0][0] == cells[2][2] &&
   cells[0][0] == symbol)
   win = true;
if (cells[0][2] == cells[1][1] &&
   cells[0][2] == cells[2][0] &&
   cells[0][2] == symbol)
   win = true;
```

```
win = checkRow(cells, symbol) ||
      checkCol(cells, symbol) ||
      checkDiagonal(cells, symbol);
```

- This does not reduce the length of code, but **easier to manage!**
- Advice: keep the length of a method within a screen height without scrolling.