Methods in Java- Defining and Using Methods

Agenda

- What is a Method?
- Naming and Best Practices
- Declaring and Invoking Methods
  - Void and Return Type Method
- Methods with parameters
- Value vs. Reference types
- Overloading Methods

What is a Method?

What is a method in Java?

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

The most important method in Java is the **main()** method.

The main() is the starting point for JVM to start execution of a Java program. Without the main() method, JVM will not execute the program.

- Named block of code, that can be invoked later
- Sample method definition:

Method named printHello

```
public static void printHello(){
  System.out.println("Hello!");
}
```

Method body always surrounded by { }

- Invoking(calling) the method several times:

```
printHello();
printHello();
```

Why Use Methods?

- More manageable programming
  - Splits large problems into small pieces
  - Better organisation of the program
  - Improves code readability
  - Improves code understandability
- Avoiding repeating code
  - Improves code maintainability
- Code reusability
  - Using existing methods several times

Naming and Best Practices

Naming Methods

- Methods naming guidelines
  - Use meaningful method naming
  - Method names should answer the question:
  - What does this method do?

findStudents,numOfStudents etc.

  - If you cannot find a good name for a method , think about whether it has a clear intent

method1,DoSOmething,Bla,firstmetho

Naming method parameters

- Method parameters names
  - Preferred form : [Noun] or [Adjectives] + [Noun]
  - Should be in camelCase
  - Should be meaningful

```
firstName, report, usersList,
 fontSizeInPixels, font, age
```

  - Unit of measure should be obvious

```
p, p1, populate,
LastName,last_name
```

Methods- Best Practices

- Each method should perform a single, well-defined task
  - A Method's name should describe that task in a clear and non-ambigious way

- Avoid methods longer than one screen
  - Split them to several shorter methods

```
private static void printReceipt() {
printHeader();
printBody();
printFooter();
}
```

Self documenting
and easy to test

Declaring and Invoking Methods

Method Declaration

In general, method declarations have 6 components:

1. Modifier
2. The return type
3. Method name
4. Parameter list
5. Exception list
6. Method body

Method Declaration components

1. Modifier: It defines the **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 types of access specifiers.

- **public:** It is accessible in all classes in your application.
- **protected:** It is accessible within the class in which it is defined and in its subclass/es
- **private:** It is accessible only within the class in which it is defined.
- **default:** It is declared/defined without using any modifier. It is accessible within the same class and package within which its class is defined.

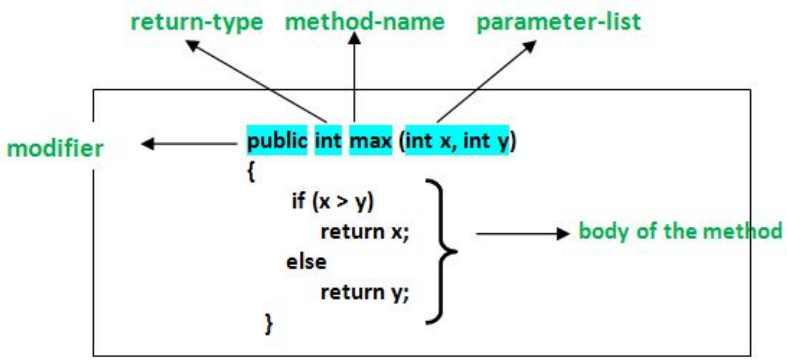
2. The return type: The data type of the value returned by the method or void if does not return a value. It is **Mandatory** in syntax.

3. Method Name: the rules for field names apply to method names as well, but the convention is a little different. It is **Mandatory** in syntax.

4. Parameter list: Comma-separated list of the input parameters is defined, preceded by their data type, within the enclosed parentheses. If there are no parameters, you must use empty parentheses (). It is **Optional** in syntax.

5. Exception list: The exceptions you expect by the method can throw, you can specify these exception(s). It is **Optional** in syntax.

6. Method body: it is enclosed between braces. The code you need to be executed to perform your intended operations. It is **Optional** in syntax.



- Methods are declared inside a class
- Variables inside a method are local

Invoking a method

- Methods are first declared , then invoked(many times)

```
public static void printBody() {
System.out.println("- - - - - - - - - - - - - ");
}
```

Method Declaration

- Methods can be invoked(called) by their name + ( ):

```
public static void main (String[ ] args) {
printBody();
}
```

Method Invocation

Invoking a method (2)

A method can be invoked from :

- The main method - main()

```
public static void main (String[ ] args) {
printBody();
}
```

- Its own body- recursion

```
static void printBody() {
printBody();
}
```

- Some other methods

```
public static void printHeader() {
printHeaderTop();
printHeaderBottom();
}
```

Break 10 minutes

Void methods

Void Method

The void keyword in Java is used to specify that a method should not return any value.

In other words, the method performs a task but does not give anything back to the caller.

When a method is declared void, it cannot return any value using the return statement.

You can only use return; by itself to exit the method early.

Example:

```
public class HelloWorld {
    public static void main(String[] args) {
        printMessage();  // calling the void method
    }

    // A method that returns no value.
    public static void printMessage() {
        System.out.println("Hello, World!");
    }
}
```

In this example, the printMessage() method has a void return type, so it doesn't return any value. It performs an action (printing a message to the console) but doesn't give anything back to the method that called it (main in this case).

Returning values from method

The return statement

In Java, the return statement is used to explicitly return a value from a method. It can also be used to exit a method early, before reaching the end.

The return statement must be compatible with the return type declared in the method signature.

For instance, if the method return type is int, then the return statement should return an integer value.

Using the return values

Assigned to a Variable

The most straightforward way to use a return value is to assign it to a variable.

```
int max = getMax(5, 10);  // Assuming getMax(int, int) returns the maximum of two integers
```

Used in an Expression

The returned value can also be directly used in an expression.

```
double total = getPrice() * quantity * 1.20;  // Assuming getPrice() returns the price of an item
```

Passed to Another Method

The returned value can be immediately passed as an argument to another method without being stored in a variable.

```
int age = Integer.parseInt(sc.nextLine());  // Here, sc.nextLine() returns a String, which is then parsed to an int
```

Methods with parameters

Method parameters

Parameters increase generality and applicability of a method:

1) Method without parameters

```
int square () { return 10*10; }
```

2) Method with parameters

```
Int square(int i) {return i * i: }
```

Parameter: a variable receiving value at the time the method is invoked

Argument: a value passed to the method when it is invoked.

Parameters vs Arguments:

- Parameter: A variable defined by a method that receives a value when the method is called. In the example int square(int i), i is a parameter.
- Argument: The actual value that is passed into the method when it is invoked. If you call square(5), then 5 is the argument.

Method parameters (2)

- You can pass zero or several parameters
- You can pass parameters of different types
- Each parameter has name and type

Multiple parameters of different types

Parameter type

Parameter name

```
public static void printStudent(String name, int age, double grade) {
System.out.println("Student: " + name + " Age: " + age + " Grade: " + grade);

 }
```

Example:

```
class Box {
double width;
 double height;
 double depth;

 double volume() {
return width * height * depth;
    }
 void setDim(double w, double h, double d) {
 width = w; height = h; depth = d;
     }
 }
```

Method parameters

Parameters increase generality and applicability of a method:

1) Method without parameters

```
int square () { return 10*10; }
```

2) Method with parameters

```
Int square(int i) {return i * i: }
```

Parameter: a variable receiving value at the time the method is invoked

Argument: a value passed to the method when it is invoked.

Parameters vs Arguments:

- Parameter: A variable defined by a method that receives a value when the method is called. In the example int square(int i), i is a parameter.
- Argument: The actual value that is passed into the method when it is invoked. If you call square(5), then 5 is the argument.

Break 10 minutes

Value vs. Reference type

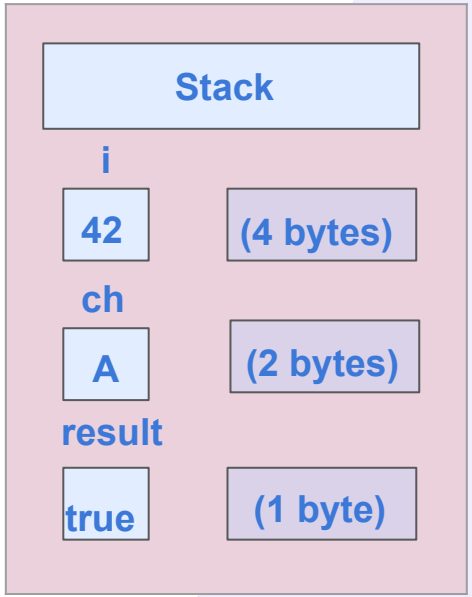
Value Type

What is a ValueType in Java?

A ValueType is a type that represents a value. This is similar to how primitive types are represented in Java. The main difference is that ValueType is a reference type, which means that it can be stored in a variable or passed as an argument to a method.

- Each variable has its own copy of the value

```
int i = 42;

char ch =”A”;

boolean result = true;
```



Reference Type

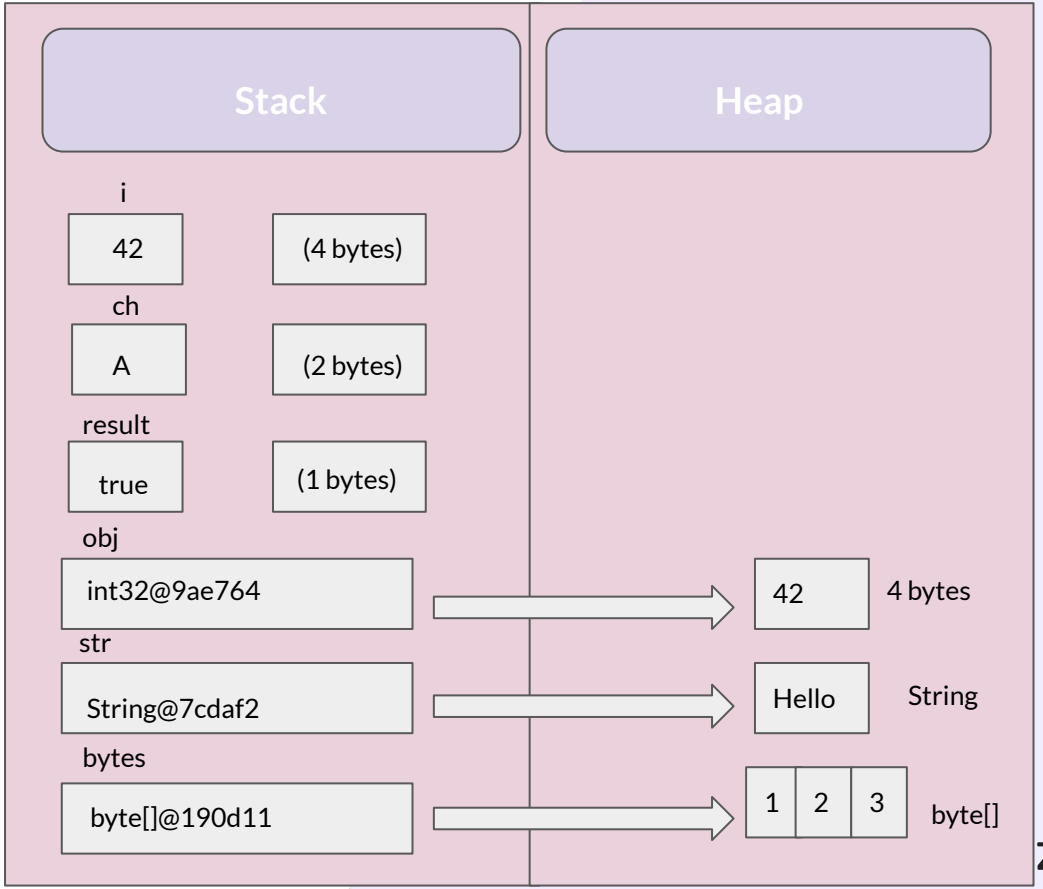
A reference data type in Java is a type of data that makes references to objects in memory rather than storing the values of those objects. It is called an object type. The reference data type stores the address of the object, which can be used to access the object's properties and methods. They include objects such as Strings, arrays, classes, and interfaces.

How to use reference data types in Java?

Using reference data types in Java involves creating objects, assigning them to variables, and accessing their properties and methods using those variables.

Value Type vs. Reference Type

```
int i = 42;

char ch = “A”;

boolean result= true;

Object obj= 42;

String str = “Hello”;

Byte[] bytes = {1 , 2 , 3};
```



Overloading Method

Method Signature

- The combination of method`s name and parameters is called signature

```
public static void print(String text){
   System.out.println(text);
}
```

Method`s signature

- Signature differentiates between methods with same names
- When methods with the same name have different signature, this is called method “overloading”

Overloading Method

- Using the same name for multiple methods with different signatures (method name and parameters)

```
public static void print(int number){
   System.out.println(number);
}
```

```
public static void print(String text){
   System.out.println(text);
}
```

```
public void print(String text, int number){
   System.out.println(text + "  " + number);
}
```

Different method signatures

START STEPS

Questions?

Thank you for your attention!