Part 5

Primitive and reference variables

Learning Objectives

- You understand the terms primitive and reference variable.
- You know the types of primitive variables in Java, and also that there can be practically an infinite number of different reference variables.
- You know the differences in behavior between primitive and reference variables when values are assigned to them, or when they're used as method parameters.

Variables in Java are classified into primitive and reference variables. From the programmer's perspective, a primitive variable's information is stored as the value of that variable, whereas a reference variable holds a reference to information related to that variable. reference variables are practically always objects in Java. Let's take a look at both of these types with the help of two examples.

```
int value = 10;
System.out.println(value);
```

Sample output

```
10
```

```
public class Name {
    private String name;

    public Name(String name) {
        this.name = name;
    }
}
```

```
Name luke = new Name("Luke");
```

```
System.out.println(luke);
```

Sample output

```
Name@4aa298b7
```

In the first example, we create a primitive `int` variable, and the number 10 is stored as its value. When we pass the variable to the `System.out.println` method, the number `10` is printed. In the second example, we create a reference variable called `luke`. A reference to an object is returned by the constructor of the `Name` class when we call it, and this reference is stored as the value of the variable. When we print the variable, we get `Name@4aa298b7` as output. What is causing this?

The method call `System.out.println` prints the value of the variable. The value of a primitive variable is concrete, whereas the value of a reference variable is a reference. When we attempt to print the value of a reference variable, the output contains the type of the variable and an identifier created for it by Java: the string `Name@4aa298b7` tells us that the given variable is of type `Name` and its identifier is `4aa298b7`.

The previous example applies whenever the programmer has not altered an object's default print format. You can modify the default print by defining the `toString` method within the class of the given object, where you specify what the objects print should look like. In the example below, we've defined the `public String toString()` method within the `Name` class, which returns the instance variable `name`. Now, when we print any object that is an instance of the `Name` class with the `System.out.println` command, the string returned by the `toString` method is what gets printed.

```
public class Name {
    private String name;

    public Name(String name) {
        this.name = name;
    }

    public String toString() {
        return this.name;
    }
}
```

```
Name luke = new Name("Luke");
System.out.println(luke); // equal to System.out.println(luke.toString());
```

Sample output

```
Luke
```

Primitive Variables

Java has eight different primitive variables. These are: `boolean` (a truth value: either `true` or `false`), `byte` (a byte containing 8 bits, between the values `-128` and `127`), `char` (a 16-bit value representing a single character), `short` (a 16-bit value that represents a small integer, between the values `-32768` and `32767`), `int` (a 32-bit value that represents a medium-sized integer, between the values -2^{31} and $2^{31}-1$), `long` (a 64-bit value that represents a large integer, between values -2^{63} and $2^{63}-1$), `float` (a floating-point number that uses 32 bits), and `double` (a floating-point number that uses 64 bits).

Out of all of these, we've mainly been using the truth value (`boolean`), integer (`int`) and floating-point variables (`double`).

```
boolean truthValue = false;
int integer = 42;
double floatingPointNumber = 4.2;

System.out.println(truthValue);
System.out.println(integer);
System.out.println(floatingPointNumber);
```

Sample output

```
false
42
4.2
```

Declaring a primitive variable causes the computer to reserve some memory where the value assigned to the variable can be stored. The size of the storage container reserved depends on type of the primitive. In the example below, we create three variables. Each one has its own memory location to which the value that is assigned is copied.

```
int first = 10;
int second = first;
int third = second;
System.out.println(first + " " + second + " " + third);
second = 5;
System.out.println(first + " " + second + " " + third);
```

```
10 10 10
10 5 10
```

The name of the variable tells the memory location where its value is stored. When you assign a value to a primitive variable with an equality sign, the value on the right side is copied to the memory location indicated by the name of the variable. For example, the statement `int first = 10` reserves a location called `first` for the variable, and then copies the value `10` into it.

Similarly, the statement `int second = first;` reserves in memory a location called `second` for the variable being created and then copies into it the value stored in the location of variable `first`.

The values of variables are also copied whenever they're used in method calls. What this means in practice is that the value of a variable that's passed as a parameter during a method call is not mutated in the calling method by the method called. In the example below, we declare a 'number' variable in the main method whose value is copied as the method call's parameter. In the method being called, the value that comes through the parameter is printed, its value is then incremented by one. The value of the variable is then printed once more, and the program execution finally returns to the main method. The value of the 'number' variable in the main method remains unaltered because it has nothing to do with the 'number' variable defined as the parameter of the method that's called.

```
1  public class Example {
2      public static void main(String[] args) {
3          int number = 1;
4          call(number);
5
6          System.out.println("Number still: " + number);
7      }
8
9      public static void call(int number) {
10         System.out.println("Number in the beginning: " + number);
11         number = number + 1;
12         System.out.println("Number in the end: " + number);
13     }
14 }
```

main:3
Name Value

Output

Reference Variables

All of the variables provided by Java (other than the eight primitive variables mentioned above) are reference type. A programmer is also free to create their own variable types by defining new classes. In practice, any object instanced from a class is a reference variable.

Let's look at the example from the beginning of the chapter where we created a variable called 'leevi' of type Name.

```
Name leevi = new Name("Leevi");
```

The call has the following parts:

- Whenever a new variable is declared, its type must be stated first. We declare a variable of type `Name` below. For the program to execute successfully, there must be a `Name` class available for the program to use.

```
Name ...
```

- We state the name of the variable as its declared. You can reference the value of the variable later on by its name. Below, the variable name is defined as leevi.

```
Name leevi...
```

- Values can be assigned to variables. Objects are created from classes by calling the class constructor. This constructor defines the values assigned to the instance variables of the object being created. We're assuming in the example below that the class `Name` has a constructor that takes a string as parameter.

```
... new Name("Leevi");
```

- The constructor call returns a value that is a reference to the newly-created object. The equality sign tells the program that the value of the right-hand side expression is to be copied as the value of the variable on the left-hand side. The reference to the newly-created object, returned by the constructor call, is copied as the value of the variable `leevi`.

```
Name leevi = new Name("Leevi");
```

The most significant difference between primitive and reference variables is that primitives (usually numbers) are immutable. The internal state of reference variables, on the other hand, can typically be mutated. This has to do with the fact that the value of a primitive variable is stored directly in the variable, whereas the value of a reference variable is a reference to the variable's data, i.e., its internal state.

Arithmetic operations, such as addition, subtraction, and multiplication can be used with primitive variables — these operations do not change the original values of the variables. Arithmetic operations create new values that can be stored in variables as needed. Conversely, the values of reference variables cannot be changed by these arithmetic expressions.

The value of a reference variable — i.e., the reference — points to a location that contains information relating to the given variable. Let's assume that we have a Person class available to us, containing an instance variable 'age'. If we've instantiated a person object from the class, we can get our hands on the age variable by following the object's reference. The value of this age variable can then be changed as needed.

Primitive and Reference Variable as Method Parameters

We mentioned earlier that the value of a primitive variable is directly stored in the variable, whereas the value of a reference variable holds a reference to an object. We also mentioned that assigning a value with the equality sign copies the value (possibly of some variable) on the right-hand side and stores it as the value of the left-hand side variable.

A similar kind of copying occurs during a method call. Regardless of whether the variable is primitive or reference type, the value passed to the method as an argument is copied for the called method to use. With primitive variables, the value of the variable is conveyed to the method. With reference variables, it's a reference.

Let's look at this in practice and assume that we have the following `Person` class available to us.

```
public class Person {
    private String name;
    private int birthYear;
```

```
    public Person(String name) {
        this.name = name;
        this.birthYear = 1970;
    }

    public int getBirthYear() {
        return this.birthYear;
    }

    public void setBirthYear(int birthYear) {
        this.birthYear = birthYear;
    }

    public String toString() {
        return this.name + " (" + this.birthYear + ")";
    }
}
```

We'll inspect the execution of the program step by step.

```
public class Example {
    public static void main(String[] args) {
        Person first = new Person("First");

        System.out.println(first);
        youthen(first);
        System.out.println(first);

        Person second = first;
        youthen(second);

        System.out.println(first);
    }

    public static void youthen(Person person) {
        person.setBirthYear(person.getBirthYear() + 1);
    }
}
```

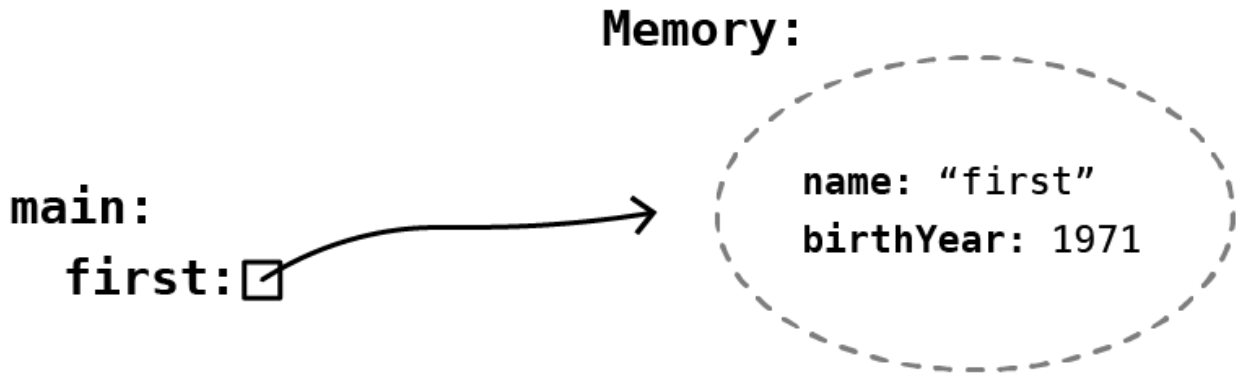
Sample output

```
First (1970)
First (1971)
First (1972)
```

The program's execution starts off from the first line of the main method. A variable of type Person is declared on its first line, and the value returned by the

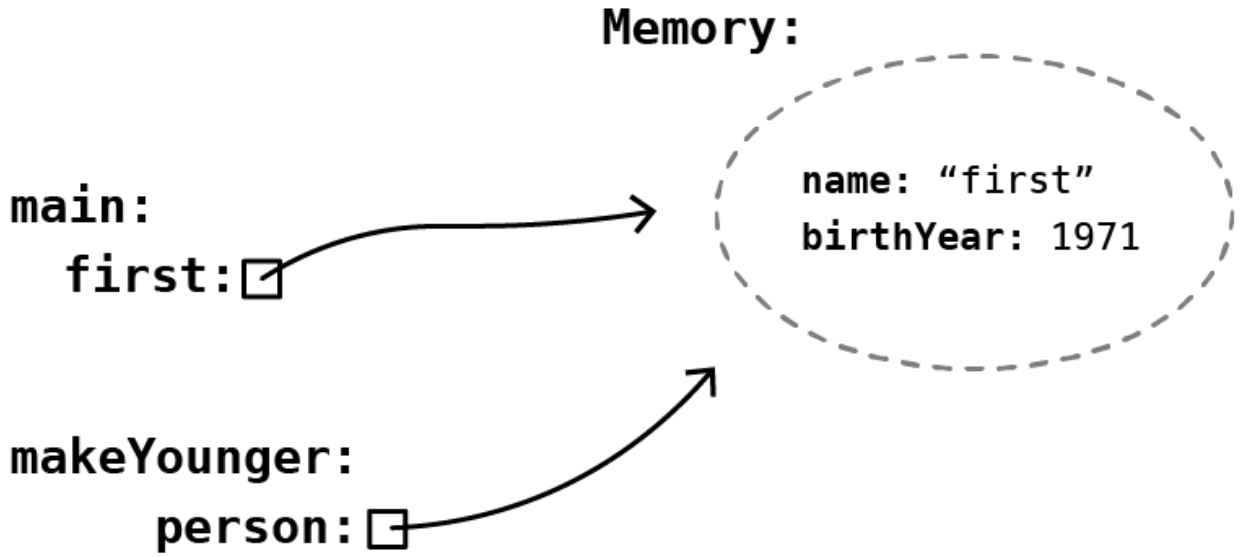
Person class constructor is copied as its value. The constructor creates an object whose birth year is set to 1970 and whose name is set to the value received as a parameter. The constructor returns a reference. Once the row has been executed, the program's state is the following — a Person object has been created in memory and the `first` variable defined in the main method contains a reference to it.

*In the illustrations below, the call stack is on the left-hand side, and the memory of the program on the right.

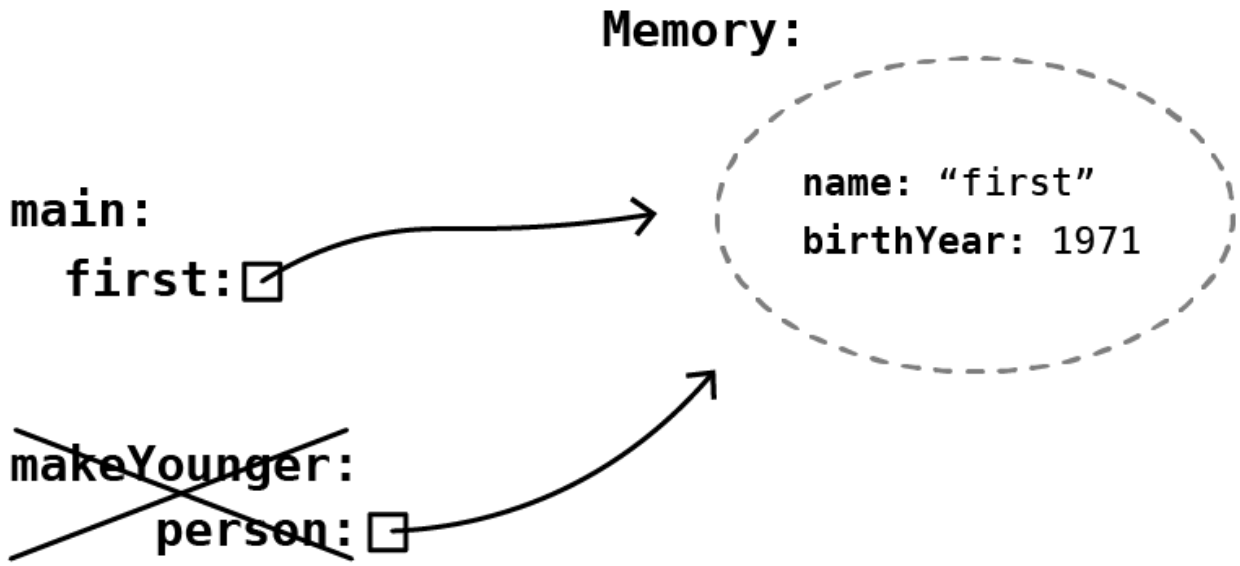


On the third row of the main method, we print the value of the variable `first`. The method call System.out.println searches for the toString method on the reference variable that has been given to it as the parameter. The Person class has the toString method, so this method is called on the object referenced by the `first` variable. The value of the `name` variable in that object is "First", and the value of the `birthYear` variable is 1970. The output becomes "First (1970)".

On the fourth row, the program calls the `youthen` method, to which we pass the variable `first` as an argument. When the method `youthen` is called, the value of the parameter variable is copied to be used by the `youthen` method. The execution of the `main` method remains waiting in the call stack. As the variable `first` is a reference type, the reference that was created earlier is copied for the method's use. At the end of the method execution, the situation is as follows — the method increments the birth year of the object it receives as a parameter by one.

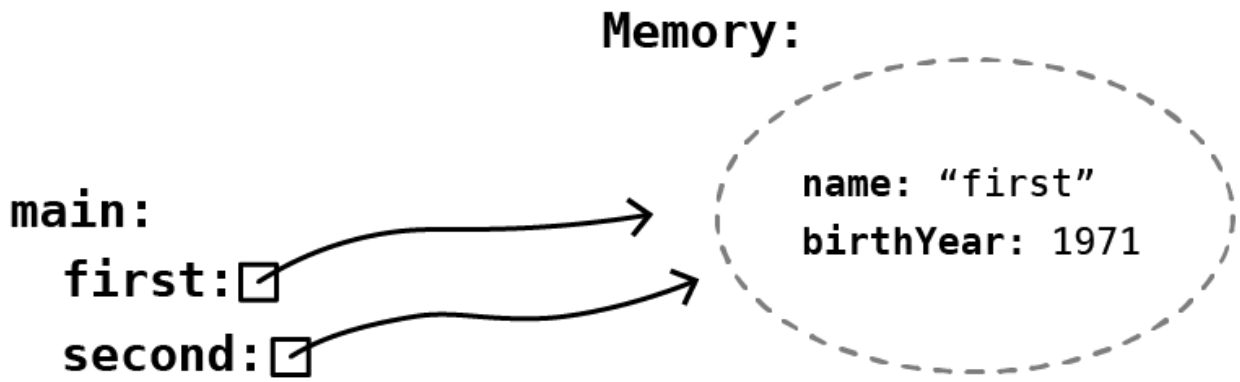


When the execution of the method makeYounger ends, we return back to the main method. The information related to the execution of the makeYounger disappears from the call stack.

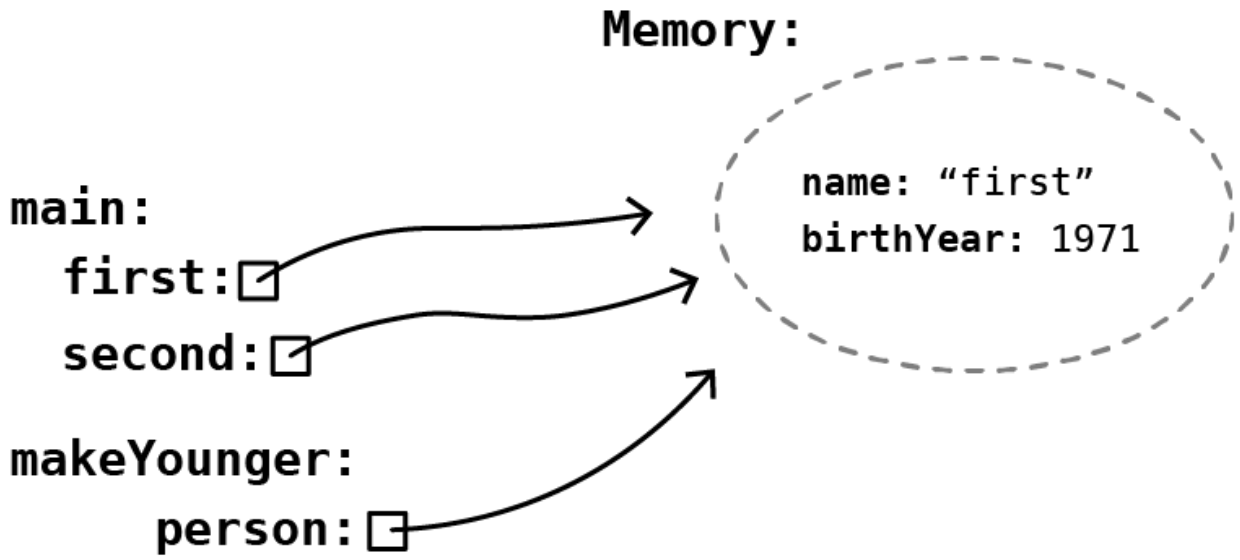


Once we've returned from the method call, we once again print the value of the variable `first`. The object referenced by the variable `first` has been mutated during the `youthen` method call: the `birthYear` variable of the object has been incremented by one. The final value printed is "First (1971)".

A new Person-type variable called `second` is then declared in the program. The value of the variable `first` is copied to the variable second, i.e., the value of the variable `second` becomes a reference to the already-existing Person object.



The program calls the `youthen` method after this, which is given the `second` variable as a parameter. The value of the variable passed during the method call is copied as a value for the method, i.e., the method receives the reference contained in the `second` variable for its use. After the method's execution, the birth year of the object referenced by the method has increased by one.



Finally, the method execution ends, and the program returns to the main method where the value of the variable first is printed one more time. The final result of the print is "First(1972)".

Quiz:
Modifying an object variable

Points:
1/1

Which number will be printed?

```
public class Counter {

    private int value;

    public Counter(int value) {
        this.value = value;
    }

    public void modifyValue(int modifier) {
        this.value = this.value + modifier;
    }

    public int getValue() {
        return value;
    }

}

--------------------------------

public static void main(String[] args) {
    Counter counter = new Counter(5);
    counter.modifyValue(8);
    System.out.println(counter.getValue());
}
```

Select the correct answer

- 3
- 5
- -3
- 8

✓ 13

The answer is correct

Answered The number of tries is not limited

Quiz:
Time machine

Points:
1/1

What will be printed?

```
public class Person {

    private int year;

    public Person() {
        this.year = 1996;
    }

    public int getYear() {
        return this.year;
    }

    public void setYear(int year) {
        this.year = year;
    }

}

------------------------------

public class TimeMachine {

    private Person traveler;

    public TimeMachine(Person person) {
        this.traveler = person;
    }

    public void travelInTime(int years) {
        this.traveler.setYear(this.traveler.getYear() + years);
    }

}
```

```
------------------------

public static void main(String[] args) {
    Person lorraine = new Person();
    TimeMachine tardis = new TimeMachine(lorraine);
    tardis.travelInTime(6);
    System.out.println(lorraine.getYear());
}
```

Select the correct answer

- 1996
- 6
- ✓ 2002
- -1990
- 1990

The answer is correct

Answered The number of tries is not limited

Variables and Computer Memory

In the course's material, concrete details related to variables and computer memory are simplified. Topics related to memory are dealt with on a level of abstraction that's suitable for learning programming. As an example, the description that the statement `int number = 5` reserves a location for the variable `number` in the memory, and copies the value 5 into it, is sufficient with regard to the learning objectives of this course.

From the perspective of the operating system, a lot more happens when the statement `int number = 5` is executed. A locker of size 32-bits is reserved in memory for the value `5`, and another one for the `number` variable. The size of the location is determined by the type of variable in question. Once

this is done, the contents of the memory location storing the value 5 are copied into the memory location of the `number` variable.

To add to the above, the `number` variable is technically not a memory location or a container. The value of the variable `number` is an address in memory — information attached to the variable about its type specifies how much data should be retrieved from its address. As an example, this is 32 bits for an integer.

We'll return to this briefly in the advanced programming course. The topic is dealt with in depth in the Computer Organization course.

You have reached the end of this section! Continue to the next section:

→ 4. Objects and references

Remember to check your points from the ball on the bottom-right corner of the material!

In this part:

Source code of the material

This course is created by the Agile Education Research -research group of the University of Helsinki.

Credits and about the material.

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

MASSIIVISET AVOIMET VERKKOKURSSIT
MASSIVE OPEN ONLINE COURSES · MOOC.FI