Introduction to Systems Programming

Class 3

2025 - Simón Gutiérrez Brida *(Based on material by Dr. Nazareno Aguirre)*

A review on types

A type defines a set of values and which operations that can be applied to those values. In Java we have primitive and non-primitive types. Examples for the first are int, float, char, boolean; examples for the later are String, Circle, TicketMachine.

A non-primitive type is defined by a Class, i.e., a Class is a type, and objects are the values for that type/Class. In the case of classes, the Class, or type, not only a set of values, it also defines the operations that can be invoked on those values.

In the last chapter …

A class defines the data (Fields), Methods/operations, and Constructors.

Fields determine the state space of all class' objects; Methods the operation that can be invoked on a particular object; and constructors define how new objects are created.

In the last chapter …

Class structure

```
public class ClassName {

  //CLASS MEMBERS

}
```

Field structure

```
private Type fieldName;

private type fieldName;
```

In the last chapter …

Class structure

```
public class ClassName {

  //CLASS MEMBERS

}
```

Constructor structure

```
public ClassName(/*ARGUMENTS*/) {
  //BODY
}
```

In the last chapter …

Class structure

```
public class ClassName {

  //CLASS MEMBERS

}
```

Method structure

```
public Type methodName(/*ARGUMENTS*/) {
  //BODY
}

public type methodName(/*ARGUMENTS*/) {
  //BODY
}
```

In the last chapter …

Arguments: a list (possibly empty), of elements with the format
type|Type **name**
separated by a comma.

int a
String author
char c

In the last chapter …

body:

- statement;
- statement; body
- { body }

statement:

- assignment
- variable declaration
- return
- method call
- If statement
  (does not need a ; at the end)

In the last chapter …

assignment: variable = expression

variable declaration:

```
Type variableName = expression
type variableName = expression
```

return:

```
return expression
```

method call:

```
obj.methodName(/*arguments*/)
```

If statement:

```
if(condition) {
  //BODY
}
```

```
if(condition) {
  //THEN-BODY
} else {
  //ELSE-BODY
}
```

The condition must be a boolean expression

In the last chapter …

An expression is a “string” of Java code that can be evaluated into a value, some examples:

- A value (a string, an int, a char, an object, etc)
- A unary expression (!true, i++, --i)
- A binary expression, i.e.: expression binary_op expression (3 + 1, a + 1, “Hello” + “ ” + “World!”, 4 * 3)
- A method call (for a method that returns a value)
- A constructor call (new ClassName(/*arguments*/))

In the last chapter …

```
public void insertMoney(int amount)
{
    if (amount > 0) {
        balance = balance + amount;
    }
    else {
        System.out.println("Use a positive amount: " + amount);
    }
}
```

This is not a good practice, the method is having two responsibilities. We now can use assertions for this

In the last chapter …

```
public class Scopy {
   private int x;

   public Scopy(int valueForX) {
      x = valueForX;
   }

   public void printScopes() {
      System.out.println(x);
      for (int x = 0; x < 10; x++) {
         System.out.println(x);
      }
      System.out.println(x);
      {
         int x = 42;
         System.out.println(x);
      }
      {
         int x = 83;
         System.out.println(x);
      }
      System.out.println(x);
   }

}
```

Let's discuss about *printScopes*, the scopes and life cycles of all Xs.

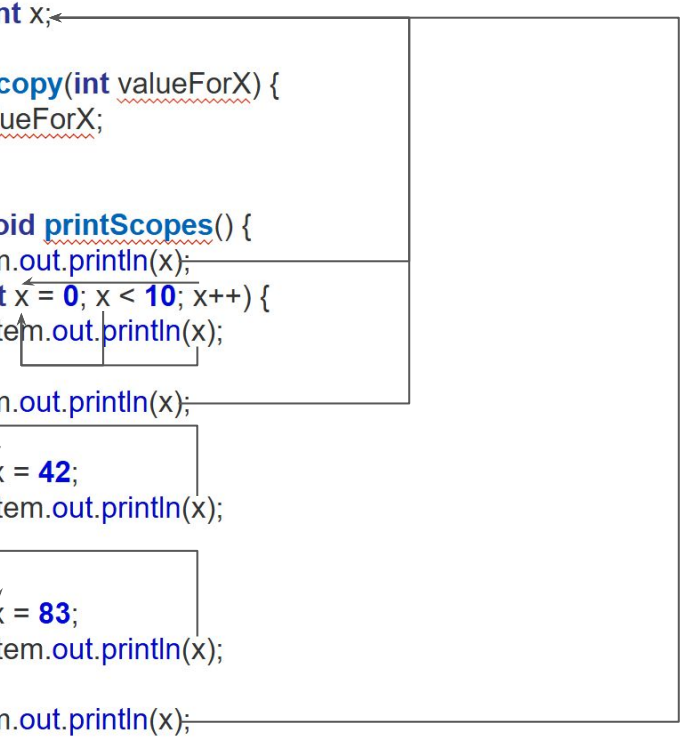
In the last chapter …

```
public class Scopy {
   private int x;

   public Scopy(int valueForX) {
      x = valueForX;
   }

   public void printScopes() {
      System.out.println(x);
      for (int x = 0; x < 10; x++) {
         System.out.println(x);
      }
      System.out.println(x);
      {
         int x = 42;
         System.out.println(x);
      }
      {
         int x = 83;
         System.out.println(x);
      }
      System.out.println(x);
   }

}
```



In the last chapter …

```
public class Scopy {
   private int x;

   public Scopy(int valueForX) {
      x = valueForX;
   }

   public void printScopes() {
      System.out.println(x);
      for (int x = 0; x < 10; x++) {
         System.out.println(x);
      }
      System.out.println(x);
      {
         int x = 42;
         System.out.println(x);
      }
      {
         int x = 83;
         System.out.println(x);
      }
      System.out.println(x);
   }

}
```

What about this?

```
public void printScopes(int x) {
   System.out.println(x);
   for (int x = 0; x < 10; x++) {
      System.out.println(x);
   }
   System.out.println(x);
   {
      int x = 42;
      System.out.println(x);
   }
   System.out.println(x);
}
```

Introduction to Systems Programming

Collections

The need to group objects/values

- Many applications involve object collections:
  - Personal organizers.
  - Library catalogs.
  - Student registration systems.

- The number of elements to store varies with the application, and many times dynamically for the application.
  - It is typically required to be able to add elements.
  - It is typically required to be able to remove elements.

Java libraries

- Groups of useful classes.
- We do not need to write everything from scratch
  - We may reuse previously developed solutions!
- Java organizes libraries in packages.
- Object grouping into collections is a recurring need in programming.
  - The *java.util* package contains classes that implement many collection implementations.

An example on collections

Let’s code for a bit
(music-organizer-v1)

Collections

- We must specify:
  - The collection kind: ArrayList.
  - The type of the objects to be stored in the collection.

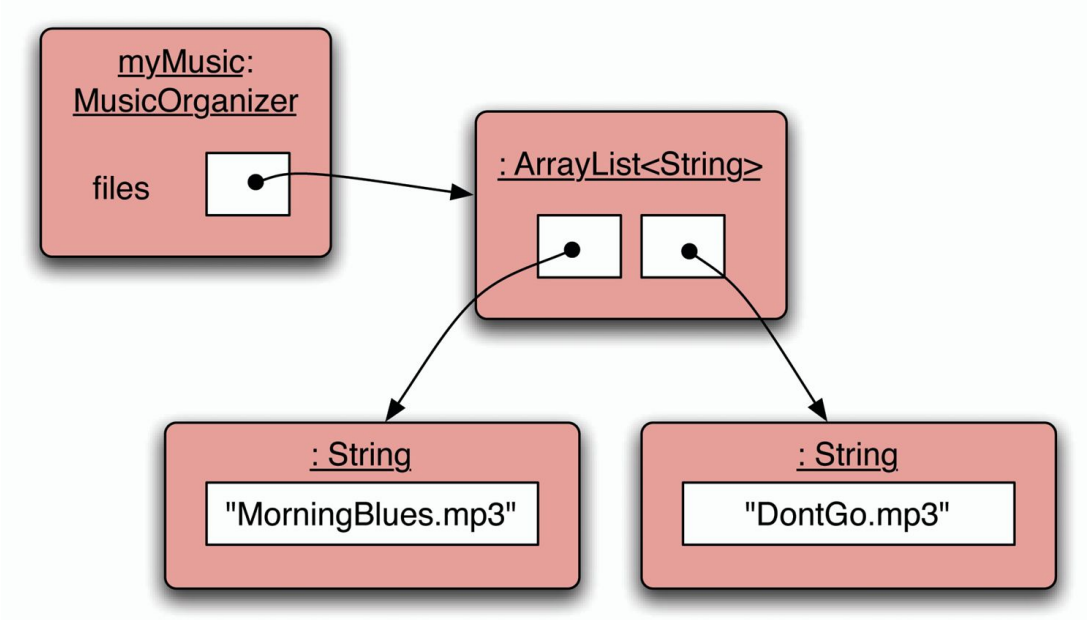
```
private ArrayList<String> files;
```

We will say this is “a list of strings” (implemented over arrays).

Generic Classes

- Collections are generic or parameterized types.
- ArrayList implements all the functionalities of a list:
  - add, get, size, etc.
- The type parameter indicates the kind of objects that we want a list to be composed of:
  - ArrayList<Person>
  - ArrayList<Integer>
  - ArrayList<String>
  - etc.

The structure of an ArrayList object



Characteristics of the list collections

- They can increment their size as needed.

- They maintain a private size counter:
  - size() is the corresponding query.

- They maintain the objects of the collection in a sequential order.
  - Each element/value has a specific index.
  - Inserting/deleting elements will affect the indexes of existing elements in the list.

- The details of how the list's functionalities are implemented are kept hidden.
  - Does it matter?
  - Can we use list collections without knowing how these are internally implemented?

Introduction to Systems Programming

Collections and iteration

Iteration

- Quite often we will want to perform certain actions an arbitrary number of times.
  - E.g., print all file names in the music organizer.
  - How many files does a music organizer have?

- The vast majority of programming languages provide us with sentences that allow us to repeat (or iterate through) actions.

- Java provides various kinds of iteration statements.
  - We will start by seeing the **for-each** iteration statement.

Iteration

- We will often need to repeat some actions multiple times.
- Iteration provides us with a mechanism to control how many times we repeat such actions.
- In the context of collections, we will often need to repeat some actions for every element in a collection.

Iteration

```
for (T elem : collection) {
    //statements to be repeated
}
```

- The collection must be of type T.
  - E.g., ArrayList<T> in this case.

- The interpretation of this statement is:
  - For every element (which we named “elem”) in the collection (which we named “collection”) execute the following statements (the “statements to be repeated”).

An example on iterations

Let's code for a bit
(in lab-classes, calculate the average credits)

Selective Processing

```
public ArrayList<String> findFiles(String searchString) {
    ArrayList<String> foundFiles = new ArrayList<String>();
    for (String filename : files) {
        if (filename.contains(searchString)) {
            foundFiles.add(filename);
        }
    }
    return foundFiles;
}
```

Through statement nesting, we can put conditional statements inside iteration statements, and get selective processing

Characteristics of For-Each Loops

- Syntactically simple, with clear semantics
- Termination of iteration occurs naturally.
- One cannot alter the collection (an error will occur if one tries to).
- We do not explicitly have an index to work with.
  - For-Each loop also work with collections that do not have ordering (do not support indexed access).
- One should not prematurely exit a for-each loop (before visiting all collection elements)
  - E.g. if we are looking for the first element that matches a condition.
  - This can be generalized to: loops should always end when the condition to loop becomes false.
- For-each loops correspond to a form of iteration known as “definite iteration”

Introduction to Systems Programming

Collections and iteration. While loops

Element search is an “indefinite” iteration

- In general, we cannot predict (before a search is performed), exactly how many objects or places we will have to traverse or visit.
- Although in many cases we may have an absolute limit (number of elements, number of positions, etc).
- Infinite iteration is a possibility.
  - Typically an error.

The While-loop

- A for-each loop repeats the loop body once per each item in a collection.
- We often need more flexibility than what the for-each loop provides.
- We can use a boolean condition to decide if we want to continue or not with the iteration.
- The while loop provides such control level.

The While-loop

```
while (booleanExpression) {
    //while's body
}
```

- The booleanExpression must be an expression of type boolean.
  - E.g., i < size.

- The interpretation of this statement is:
  - While the condition (which we named “booleanExpression”) is true, execute the following statements (the “while’s body”).

The While-loop

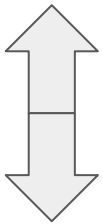
- A for-each loop repeats the loop body once per each item in a collection.
- We often need more flexibility than what the for-each loop provides.
- We can use a boolean condition to decide if we want to continue or not with the iteration.
- The while loop provides such control level.

The While-loop, an example

```
public int div(int a, int b) {
    assert a >= 0 : "a must be positive";
    assert b > 0 : "b must be greater than zero";
    int res = 0;
    while (a >= b) {
        a = a - b;
        res = res + 1;
    }
    return res;
}
```

For-Each and While

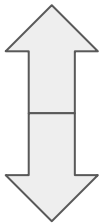
```
for (T elem : collection) {
    doSomething(elem);
}
```



```
int index = 0;
while (index < getSize(collection)) {
    T elem = getElemAt(collection, index);
    doSomething(elem);
    index++;
}
```

For-Each and While, using ArrayList

```
for (T elem : collection) {
    doSomething(elem);
}
```



```
int index = 0;
while (index < collection.size()) {
    T elem = collection.get(index);
    doSomething(elem);
    index++;
}
```

for-each versus while

- For-each:
  - Simpler and easier to write.
  - Safer: termination is guaranteed.

- While:
  - It's not limited to collections.
  - We have to be careful: we may end up having infinite loops.

Search in a collection

- A fundamental task.
- From the point of view of iteration, it's inherently "indefinite".
- We have to consider both success (element found) as failure (search is exhausted).
- Both success and failure are cases that must make the loop condition false.
- Remember that the collection may be empty!

Search in a collection

```
int index = 0;
boolean found = false;
while(index < files.size() && !found) {
    String file = files.get(index);
    if(file.contains(searchString)) {
        // We don't need to keep looking.
        found = true;
    }
    else {
        index++;
    }
}
// Either we found it at index,
// or we searched the whole collection.
```

While-loop does not need collections!

```
public boolean isPrime(int value) {
    assert value >= 0 : "value must be positive";
    int divisors = 0;
    int divisor = 1;
    while (divisor <= value) {
        if (value % divisor == 0) {
            divisors++;
        }
        divisor++;
    }
    return divisors == 2;
}
```

Java Code, statements update

body:

- statement;
- statement; body
- { body }

statement:

- assignment
- variable declaration
- return
- method call
- If statement
  (does not need a ; at the end)
- For-each
  (does not need a ; at the end)
- While
  (does not need a ; at the end)