CORE JAVA GROOMING

Responsibilities of java compiler

1. Check syntax () [] ; , . {}
2. Check semantics (types)

int a = “Hello” ;

1. Adds instruction

int a = 100 ;

double d = a ; // your point of view

double d = (double) a ; // compiler pov

1. Java code 🡪 byte code
2. Generate class file

Compile : javac ClassName.java

Execute: java ClassName

ClassName == FileName

TOKENS

Smallest unit of java program

1. Keyword – predefined words
2. Identifiers – name – variables, methods, class, package, interface, enums
   1. Rules
      1. Cannot start with number
      2. No spaces
      3. No keyword
      4. No special character(except $ and \_ )
   2. Conventions – IT standards
      1. Class/Interfces – UpperCamelCasing
      2. Variables/methods/packages - lowerCamelCasing
3. Literals – data / value
   1. Primitive – number (integer and decimals) , char – ‘a’ , boolean – true,false
   2. Non-Primitive – String
4. Separators {} [] () ; , .
5. Comments // /\* \*/
6. Operators

DATATYPES AND VARIABLES

Int a = 5 ;

1. Primitive

1byte 2 4 8 4 8

Byte < short < int < long < float < double

Char<

Boolean = 1 bit

1. Non-primitive – String, Objects, Arrays

Scope of variable

1. Local variables
2. Global variables
   1. Static – that belong to class – single copy – declared using static keyword

ClassName.variable ;

* 1. Non-static – belong to object – multiple copies

ObjectRef.variable ;

TYPECASTING

1. Primitive – widening and narrowing
2. Non-primitive – upcasting and downcasting

=====🡺 WIDENING

Byte < short < int < long < float < double

Char<

<====== NARROWING

TYPECAST OPERATOR

(newDatatype) var / value / expressions ;

Int a = 1000 ;

Double d = a ; // widening

Double d = 10.47 ;

Float f = (float) d ; // narrowing

OPERATORS

Predefined symbols – specific task

Operands 🡺 values on which we perform specific task

Based on number of operands:

1. Unary – only one operand
2. Binary – 2 operands
3. Ternary – 3 operands

Types:

1. Arithmetic 🡪 + - \* / %
2. Assignment 🡪 = += -= \*= /=

Int a = 10 ;

a += 10 ; // 🡺 a = a + 10 ;

Sopln(a) ;

1. Conditional

Operand1 ? operand2 : operand3 ;

Operand1 🡺 always boolean

Op2 🡺 when true

Op3 🡺 when false

1. Relational > < >= <= !=
2. Logical && || !
3. Increment / Decrement ++ / --

A++ 🡺 post 🡺 first use, then update

++a 🡺 pre 🡺 first update, then use

DECISION MAKING STATEMENT

1. If (Condition ) { 🡺 ONE CONDITION

// statements

}

1. If – else 🡺 TWO CONDITIONS

If (condition) {

// statement 1

}

Else {

// statement 2

}

1. Else – if ladder 🡺 MULTIPLE CONDITIONS

If (condition1) {}

Else if (condition2) {}

Else if (condition3) {}

Else {

// statement

}

1. NESTED IF – CONDITION INSIDE CONDITION

BLOOD DONATION CAMP

18 +

55 +

If (age > 18 ) {

If (weight > 55 ) {

Sopln (“Eligible for blood donation”) ;

}

}

1. Switch case (tv remotes)

Switch (values) {

Case 1 : // st1

Break ;

Case 2 : // st2

Break;

Case 3 : // st3

Break ;

Default:

}

LOOPS

1. While loop

While (condition ) {

// statements to be repeated

// updation 🡪 to terminate

}

1. Do-while loop

Do {

// statements to be repeated

// updation 🡪 To terminate

} while (condition);

1. For – loop

For (initialization ; condition ; updation) {

// statements

}

METHODS

[ACCESSSPECIFIER SPECIFIER] RETURNTYPE METHODNAME ([ARGS]) {

// instructions

}

Access Modifier 🡺 who can access the method == public, private, protected, default

Modifier 🡺 how can you access the method == static, abstract, final

Type:

1. No returntype, no arguments

Void add1 () {

Sopln(10+20 )

}

1. No returntype, with arguments

Void add2 (int a, int b) {

Sopln(a+b) ;

}

1. With returntype, no argument

int add3 () {

Int a = 10 ;

Int b = 100 ;

return a + b ;

}

1. With returntype, with argument

Int add4 (int a , int b ) {

Return a + b ;

}

* Whenever a method has return type, other than void, you can print it
* Return statement should be the last statement

Scanner class

* Dynamic reading
* Import java.util.Scanner
* Scanner inp = new Scanner(System.in) ;

Int 🡺 inp.nextInt() ;

Double 🡺 nextDouble () ;

String 🡺 next() 🡺 one word input

== nextLine() 🡺 more than one word

OBJECTS

Anything which has physical existence 🡺 real world entities

In java: block of memory

Syntax:

ClassName var = new ClassName() ;

Where is the object created? 🡺 HEAP AREA

* ALL THE OBJECTS will have states and behaviors
  + States = variables
  + Behaviours = methods

class Student {

String name = “Tom” ;

void eat () {

Sopln (“eating”) ;

}

}

Class Demo {

public static void main(String[]args) {

Student s1 = new Student() ;

Sopln(s1.name) ;

S1.eat() ;

}

}

BLOCKS

* Also called initializers
* Startup instructions before main task

Types:

1. Static blocks 🡺 executed before main method

static {

// instructions

}

1. Non-static block 🡺 during object creation

{

// instructions

}

* Where are block created? 🡺 inside class
* Executed sequentially 🡺 top to bottom

CONSTRUCTORS

* SET of instructions to initialize (assign some values) and instantiate (object creation)
* Use? To load all non static properties into the object
* ClassName == ConstructorName (rule)
* Constructors are special methods with no – returtype
* Syntax:

AccessSpecifier ClassName (optional arguments) {

// instructions

}

* Types:

1. No arguments
2. With arguments

* DEFAULT CONSTRUCTOR 🡺 if the programmer fails to create a constructor, the compiler will generate a no argument constructor that is default constructor
* This keyword
  + Reference of current working object

class Student {

String name ;

Student(String name ) {

this.name = name ;

}

}

INHERITANCE

Process of acquiring properties of parents/super class into child/sub class

Type:

1. Single level – one super class, one sub class

A 🡨 B

1. Multilevel inheritance – combination of more than one super and sub class but in sequential manner

A 🡨 B 🡨 C

1. Hierarchical inheritance – one super class having multiple sub classes

A

B C D

1. Multiple Inheritance

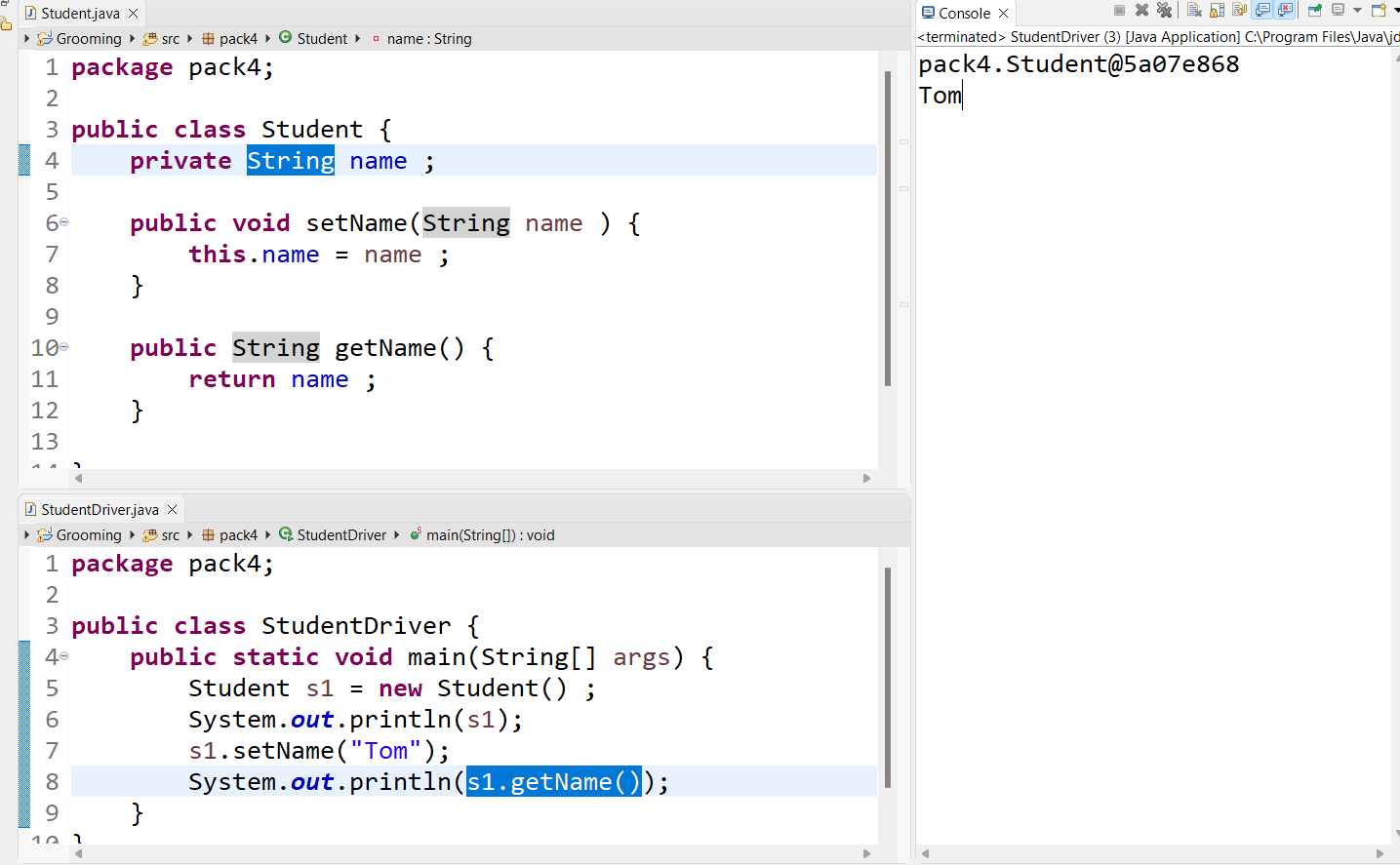
One subclass having multiple super classes 🡺 NOT POSSIBLE

1. Hybrid inheritance 🡪 combination of different types of inheritance

ENCAPSULATION

* Process of binding states and behaviors of an object
* We can achieve data hiding 🡺 instead of giving direct access to the user, we provide indirect access/safe access/controlled access to the uder
* Steps:

1. Make data members (variables) private
2. Create getter and setter methods



POLYMORPHISM

Poly = many

Morph = forms

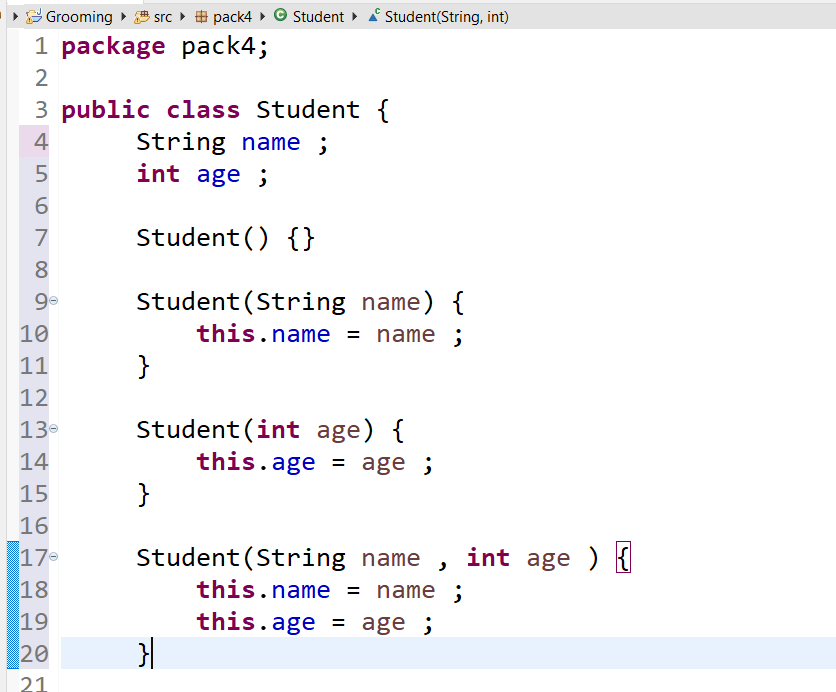
* An object exhibiting more than one behaviours

Types:

1. Compile Time 🡺 binding is achieved at compile time

Method overloading 🡺 if a class contains more than one method with same name but different arguments

Constructor overloading 🡺 if a class contains more than one constructor with different arguments



1. Run-Time 🡺 binding is achieved at run Time

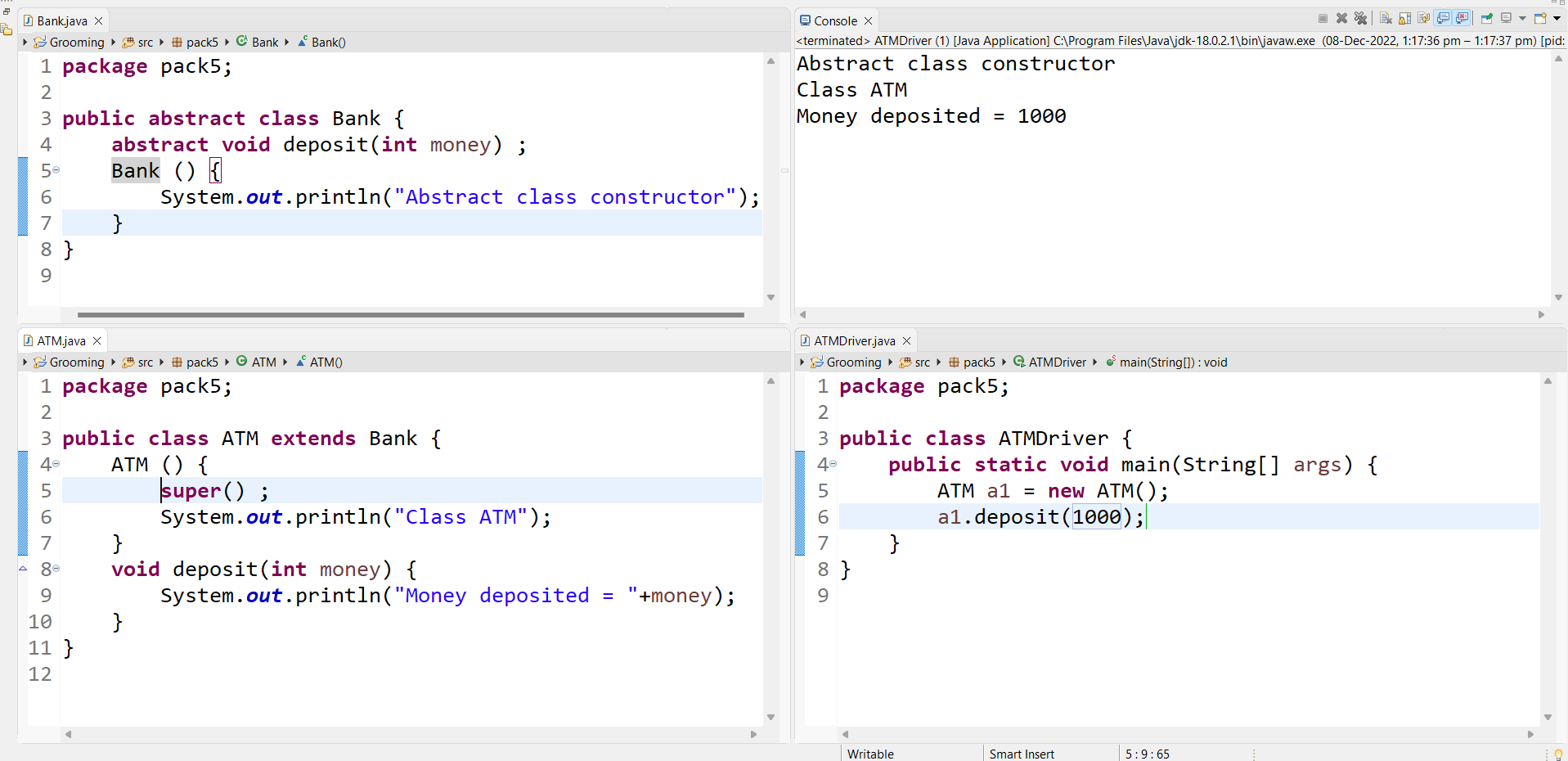
Method overriding 🡺 giving new implementation to the super class method in the sub class

ABSTRACTION

* Hiding implementation and giving only specification (only important features)
* How? Using abstract keyword
* Abstract 🡺 can be used for both methods and also classes
* Using abstract class we cannot achieve 100 % abstraction
* 100 % abstraction can be achieved using interfaces

Abstract class

1. We can create both abstract method and concrete methods
2. We cannot create objects for abstract classes
3. We can create constructors in abstract classes



1. We can create static as well as non static variables

INTERFACES

* We can create only abstract methods
* We can create concrete methods but they should be static
* We cannot create objects for interfaces
* We cannot create constructors in interfaces
* Variable 🡺 public, static, final
* Method 🡺 public, abstract

In abstraction

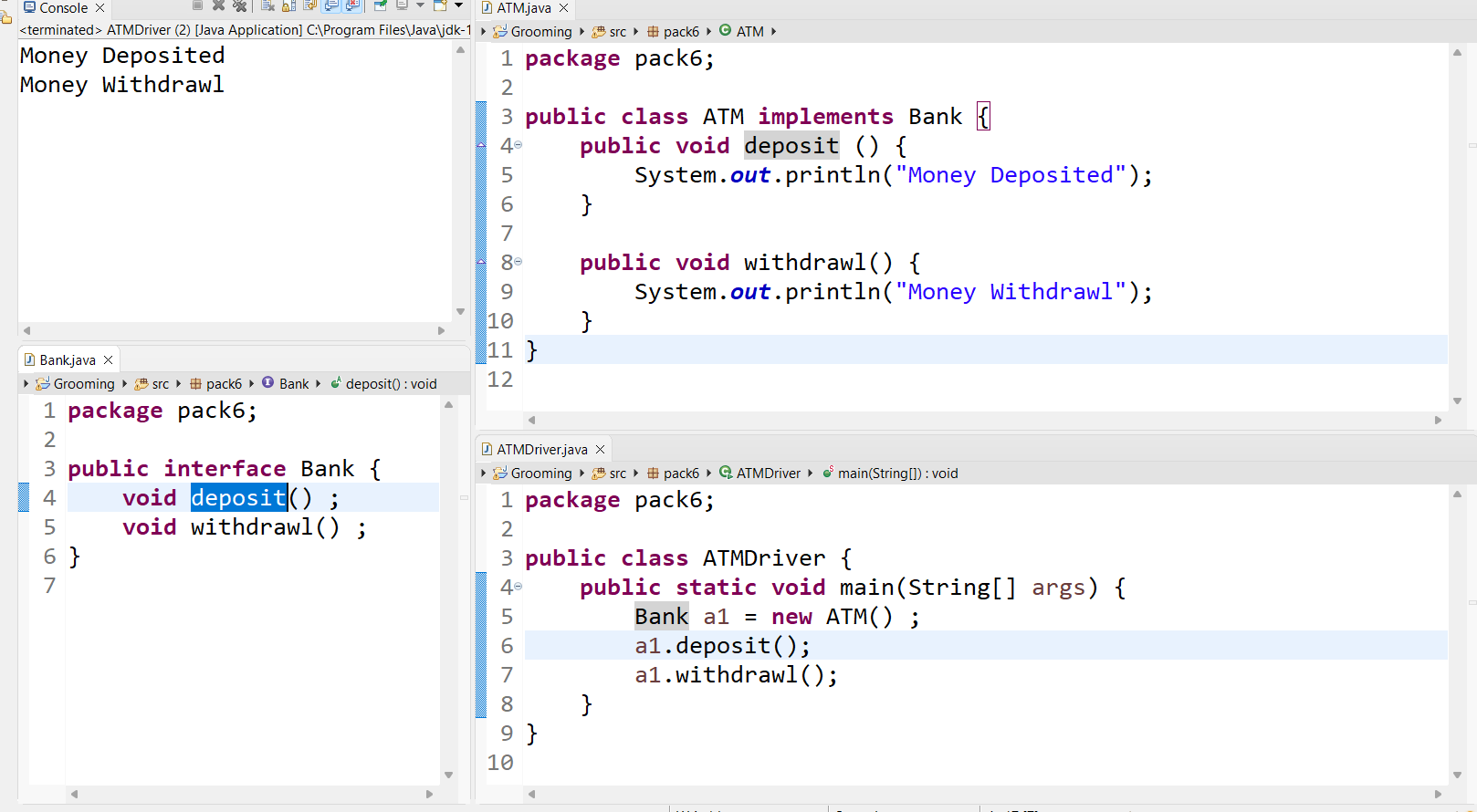
WHAT IS HAPPENING?

HOW IT IS HAPPENING? 🡺 we don’t care

With the help of implements keyword

Solves 2 problems:

1. Multiple inheritance can be achieved using interfaces
2. 100 % abstraction



OBJECT CLASS

1. IT is supermost class of all the classes
2. Present in java.lang package
3. 11 non-static methods
4. 3 important methods :
   1. toString() 🡪 String
   2. equals() 🡪 boolean
   3. hashCode() 🡪 int

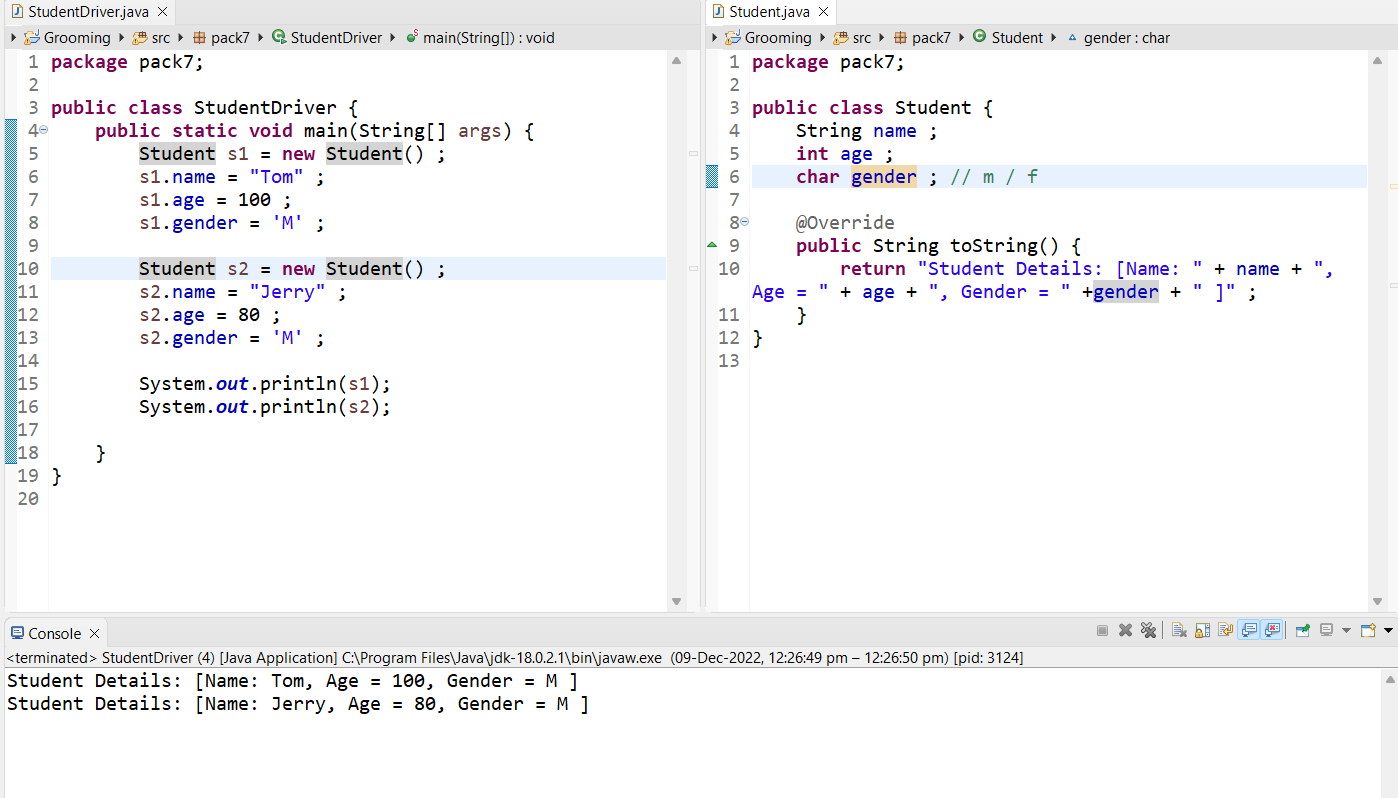
🡺 when is toString() method called ?

Whenever we print object reference variable

* WHY DO WE NEED TO OVERRIDE toString() method ?

Before overriding if we print object reference variable 🡺 address will be printed

After overriding if we print object reference 🡺 states will be printed



* WHY DO WE NEED TO OVERRIDE equals() method?

Before overriding 🡺 compare on the basis of objects

After overriding 🡺 compare on the basis of states

STRINGS

* Anything within “ ”
* “hello world”
* Present in java.lang package
* Non-primitive datatype
* String are immutable
  + Because once string object is created we cannot modify or update it.
  + When we try to modify string object A NEW STRING OBJECT will be created.
* Disadvantage: MEMORY CONSUMED IS MORE

Sushant 🡺 tnahsuS 🡺 7 objects will be created

Solutions:

1. StringBuilder
2. StringBuffer

StringBuffer

1. In StringBuffer class all methods are synchronized
2. It means it is thread safe
3. It is slower than StringBuilder

StringBuffer sb1 = new StringBuffer(“Hello World”) ;

StringBuilder

1. In StringBuilder class all methods are non-synchronised
2. It is not thread safe
3. It is faster compared to StringBuffer

StringBuilder sb2 = new StringBuilder(“Hello” ) ;

ARRAYS

Variable 🡺 one address 🡺 only one value

Arrays 🡺 one address 🡺 multiple datas

1. It is a continuous block of memory
2. It does not have a name, it has address
3. Access the elements using index
4. Multiple values can be stored 🡺 but only HOMOGENEOUS (same type)
5. Array will have fixed size, we cannot modify them later

Array Creation

1. datatype [ ] variable = new datatype [size ] ;

int [] arr = new int [ 5 ] ;

1. datatype [] variable = { element1, el2, el3 ……….. } ;

int [ ] arr2 = { 1,2,3,4,5,6} ;

how to get size of the array? 🡺 length variable

index always starts from 0

how to access the elements?

1. For loop

for ( int i = 0 ; i<arr.length ; i++ ) {

Sopln(arr [ i ] ) ;

}

1. For – each loop

For (datatype element : arrayVariable) {

Sopln (element) ;

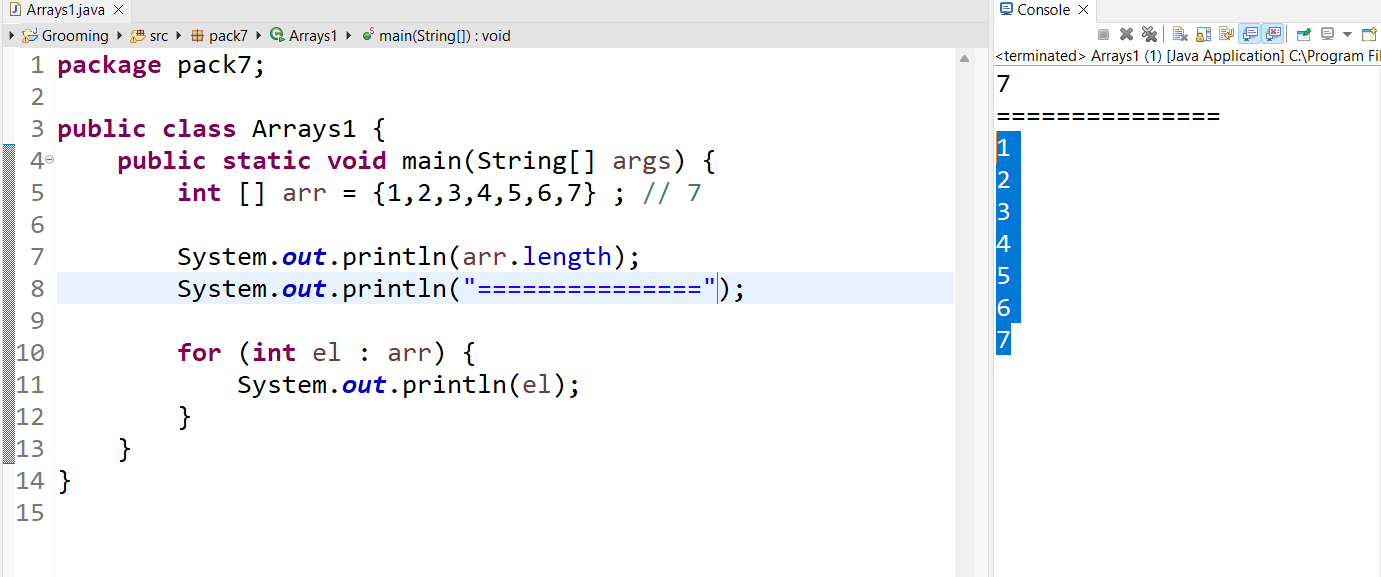
}

Int [] arr = {1,2,3,4,5,6,7} ;

for ( int el : arr ) {

Sopln(el) ;

}



EXCEPTIONS

1. Problems during runtime
2. If exceptions occur in program, program will stop abruptly

Types:

1. Checked exception
   1. Compiler aware exceptions
   2. Ex: InterruptedException, FileNotFoundException
   3. Mandatory to handle them
2. Unchecked exception
   1. Compiler unaware exceptions
   2. Ex: ArithmeticException, ArrayIndexOutOfBoundsException
   3. Warning: either handle or declare

How to handle exception?

* Using try – catch block
* Syntax:

try {

// statement which might throw exception

}

Catch(Exception e) {

// statements

}

What is the use of multiple try-catch block?

* To catch multiple exceptions

How do we declare exceptions?

* Using throw keyword

Throws 🡺 declare exception

Throw 🡺 create exception

Finally block

If the exception occurs or not occurs, finally block will execute

* RunTime exceptions and its subclasses are UNCHECKED EXCEPTIONS
* ALL other exceptions are checked exceptions

Top most class of all exceptions 🡺 Throwable class

COLLECTIONS

Int a = 10 ;

Int [] arr = {1,2,3,4,5,6} ;

Disadvantages of array

1. Size could not be modified
2. We cannot store heterogeneous data (different types of data)
3. Array manipulation 🡺 required complex logics

Hence we moved to collections

What are collections?

1. It is an interface 🡺 java.util package
2. Collection represents 🡺 group of objects

Collection 🡺 Shopping bad 🡺 heterogeneous collection

1. Methods 🡺 add, remove, access, search, count, sort elements

On data structures like Set, Queue, List

List interface

1. Child of Collection interface
2. ArrayList, LinkedList, Vector are concrete implementing classes of List interface

ArrayList

1. Implementing class of List interface
2. It will have index
3. Store duplicate data
4. Insertion order will be maintained

Set interface

1. Child of collection interface
2. Unordered collection 🡺 no index
3. Insertion order is not maintained
4. Duplicate elements are not allowed
5. HashSet and TreeSet are the concrete implementing classes of Set Interface

HashSet

1. Implanting class of Set interface
2. It does not allow duplicate elements
3. We can store only one null value
4. We can store heterogeneous types of data
5. We cannot use for loop to access the elements
6. For each loop or iterator method to access the elements of HashSet

treeSet

1. Implanting class of Set Interface
2. We cannot store null values
3. Homogeneous type of data can only be stored
4. All the elements are automatically sorted 🡪 ascending order
5. It does not allow duplicate elemets

QUEUE Interface

1. Present in java.util package
2. Extends collection interface
3. It follows 🡺 FIRST IN FIRST OUT concept

Insertion always in the end, removal at the beginning

1. Insertion order is maintained
2. Implementing classe 🡺 PriorityQueue, LinkedList

LinkedList

Engine == || Boggey1 || == || Boggey2 || == || Boggey3 || ==

Node1 Node2

Head || == || value | address|| == || value |address|| ==

ArrayList

Var | 0x1 | 🡪 | 0 | 10|12|567|23 |100|