ORACLE® ACADEMY

Java Programming

1-1
Fundamentals of Java - What I should know

ORACLE ACADEMY

Objectives

This lesson covers the following topics:

- Review of Java Primitives
- Review of Strings
- Review of Logical and Relational Operators
- Review of Conditional Statements
- Review of Program Control
- Review of Object Classes
- Review of Constructor and Method Overloading
- Review of Inheritance

Primitive Data Types

- Java has eight primitive data types that are used to store data during a program's operation.
- Primitive data types are a special group of data types that do not use the keyword new when initialized.
- Java creates them as automatic variables that are not references, which are stored in memory with the name of the variable.
- The most common primitive types used in this course are int (integers) and double (decimals).

Primitive Data Types

Data Type	Size	Example Data	Data Description
boolean	1 bit	true, false	true, false
byte	1 byte (8 bits)	12, 128	Stores integers from -128 to 127
char	2 bytes	'A', '5', '#'	Stores a 16-bit Unicode character
short	2 bytes	6, -14, 2345	Stores integers from -32,768 to 32,767.

Primitive Data Types

Data Type	Size	Example Data	Data Description
int	4 bytes	6, -14, 2345	Stores integers from: -2,147,483,648 to 2,147,483,647
long	8 bytes	3459111, 2	Stores integers from: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	3.145, .077	Stores a positive or negative decimal number from: 1.4023x10^{-45} to 3.4028x10^{+38}
double	8 bytes	.0000456, 3.7	Stores a positive or negative decimal number from: 4.9406x10^{-324} to 1.7977x10^{+308}

Declaring Variables and Using Literals

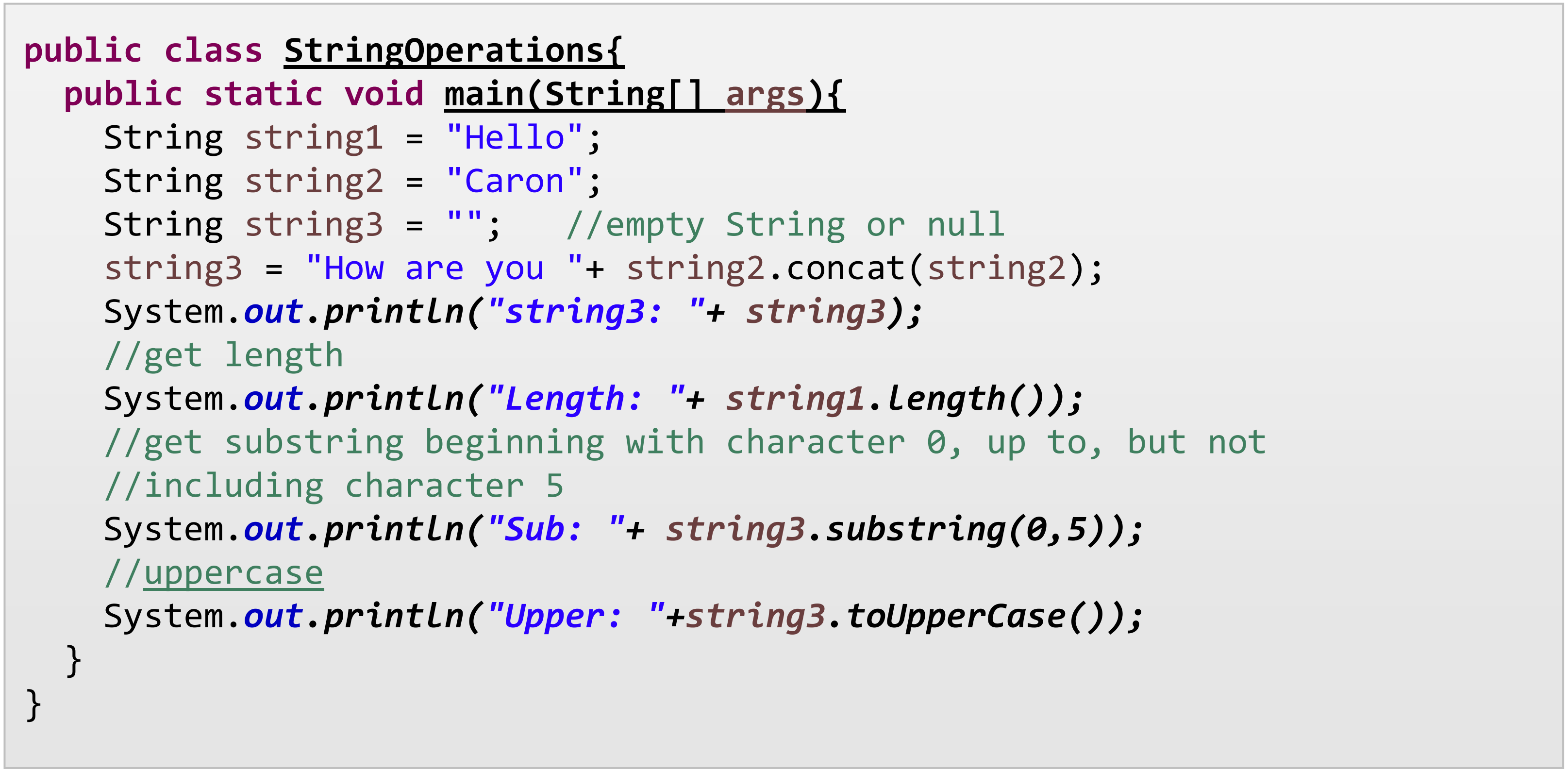
- The keyword new is not used when initializing a variable primitive type.
- Instead, a literal value should be assigned to each variable upon initialization.
- A literal can be any number, text, or other information that represents a value.
- Examples of declaring a variable and assigning it a literal value:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
long creditCardNumber = 1234_5678_9012_3456L;
```

Strings

- A String is an object that contains a sequence of characters.
- Declaring and instantiating a String is much like any other object variable.
- However, there are differences:
  - They can be instantiated (created) without using the new keyword.
  - They are immutable.
  - Once instantiated, they are final and cannot be changed.

String Operations Example



```
public class StringOperations{
  public static void main(String[] args){
    String string1 = "Hello";
    String string2 = "Caron";
    String string3 = "";    //empty String or null
    string3 = "How are you "+ string2.concat(string2);
    System.out.println("string3: "+ string3);
    //get length
    System.out.println("Length: "+ string1.length());
    //get substring beginning with character 0, up to, but not
    //including character 5
    System.out.println("Sub: "+ string3.substring(0,5));
    //uppercase
    System.out.println("Upper: "+string3.toUpperCase());
  }
}
```

compareTo Method

- There are methods to use when comparing Strings.
- Method: s1.compareTo(s2);
- Should be used when trying to find the lexicographical order of two strings.
- Returns an integer.
  - If s1 is less than s2, an int < 0 is returned.
  - If s1 is equal to s2, 0 is returned.
  - If s1 is larger than s2, an int > 0 is returned.

equals Method

- Method: s1.equals(s2)
- Should be used when you only wish to find if the two strings are equal.
- Returns a boolean value.
  - If true is returned, s1 is equal to s2
  - If false is returned, s1 is not equal to s2.

Scanner

- To read in the input that the user has entered, use the Java object Scanner. You will have to use an import statement to access the class java.util.Scanner

```java
import java.util.Scanner;
```

- To initialize a Scanner, write:

```java
Scanner in = new Scanner(System.in);
```

- To read the next string from the console:

```java
String input = in.next();
```

- To read the next integer:

```java
int answer = in.nextInt();
```

Relational Operators

- Java has six relational operators used to test primitive or literal numerical values.
- Relational operators are used to evaluate if-else and loop conditions.

Relational Operator	Definition
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

Logic Operators

- Java has three logic operators used to combine boolean expressions into complex tests.

Logic Operator	Meaning
&&	And
||	Or
!	Not

If Conditional Statement

- To build an if-else statement, remember the following rules:
  - An if-else statement needs a condition or method that is tested for true/false.
- For example:
  - if(x==5)
  - if(y >= 17)
  - if(s1.equals(s2))

if-else Statements with the int Data Type

```java
import java.util.Scanner;
public class ValueChecker{
  public static void main(String[] args){
    Scanner in = new Scanner(System.in);
    int value = 0;
    System.out.println("Enter a number:");
    value = in.nextInt();
    if( value == 7) {
      System.out.println("That's lucky!");
    }
    else if( value == 13) {
      System.out.println("That's unlucky!");
    }
    else {
      System.out.println("That is neither lucky nor unlucky!");
    }
  }
}
```

While Loop

- With a while loop, Java uses the syntax:

```
while(condition is true){
  //logic
}
```

- Similar to if statements, the while loop parameters can be boolean types or can equate to a boolean value.
- Conditional statements (<, >, <=, >=, !=, ==) equate to boolean values.
- Examples:
  - while (num1 < num2)
  - while (isTrue)
  - while (n !=0)

The do-while Loop

- The do-while loop:
  - Is a post-test loop.
  - Is a modified while loop that allows the program to run through the loop once before testing the boolean condition.
  - Continues until the condition becomes false.
- If you do not allow for a change in the condition, the loop will run forever as an infinite loop.

```
do{
   //statements to repeat go here
} while(condition);
```

The for Loop

- The for loop repeats code a pre set number of times.
- for loop syntax contains three parts:
- Initializing the loop counter.
- Conditional statement, or stopping condition.
- Updating the counter (going to the next value).
  - Think of i as a counter starting at 0 and incrementing until i=timesToRun.

```java
for(int i=0; i < timesToRun; i++){
  //logic
}
```

Array

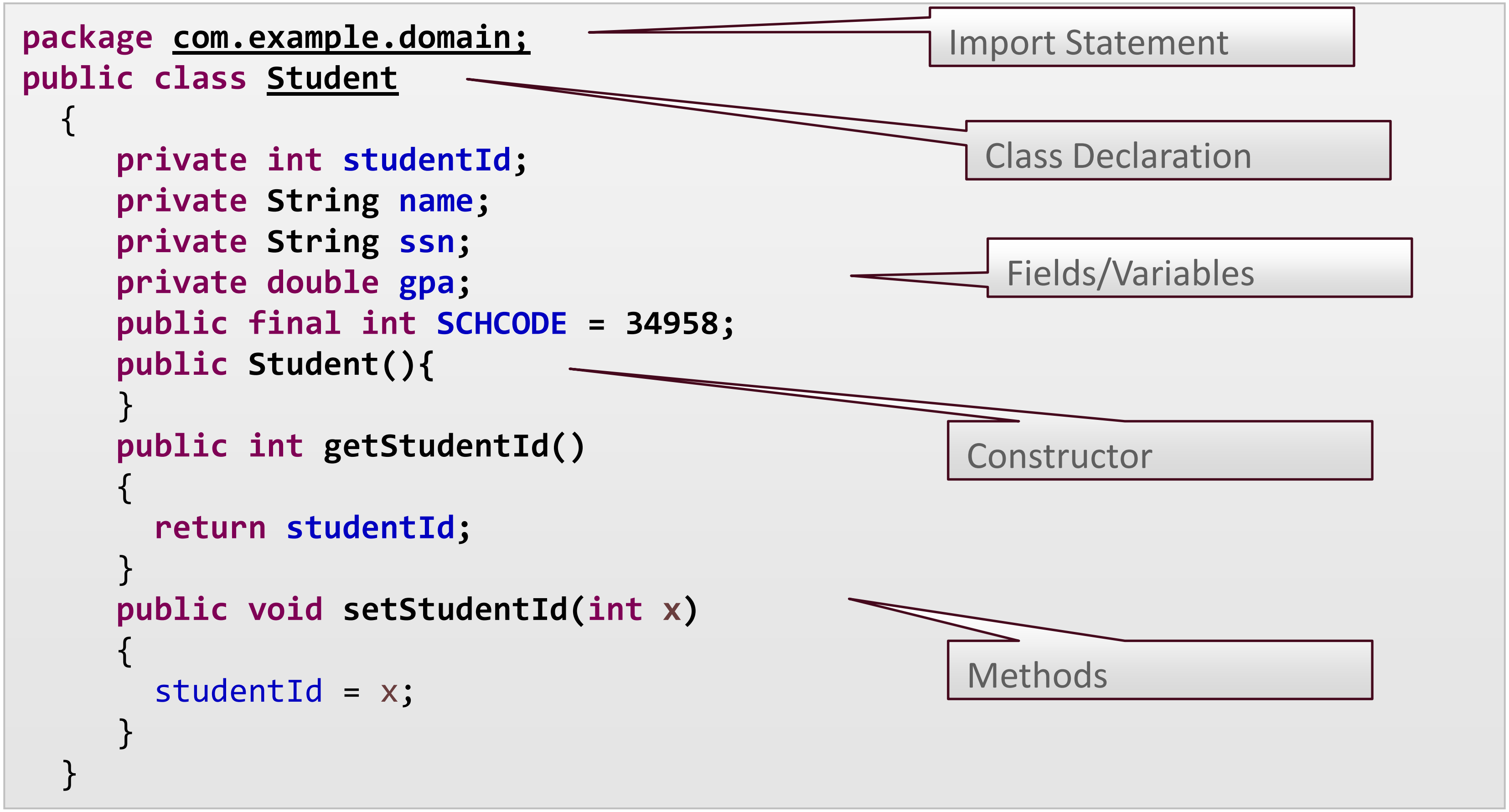
- An array is a collection of values of the same data type stored in a container object.
- Can be any number of values.
- Length of the array is set when the array is declared.
- Size is fixed once the array is declared.
- Array examples:

```
String[] myBouquet = new String[6];
int[] myArray = {7, 24, 352, 2, 37};
```

Object Classes

- Object classes:
  - Are classes that define objects to be used in a driver class.
  - Can be found in the Java API, or created by you.
  - Examples: String, BankAccount, Student, Rectangle

Student Class Example



```
package com.example.domain;
public class Student
  {
     private int studentId;
     private String name;
     private String ssn;
     private double gpa;
     public final int SCHCODE = 34958;
     public Student(){
     }
     public int getStudentId()
     {
       return studentId;
     }
     public void setStudentId(int x)
     {
       studentId = x;
     }
  }
```

Import Statement

Class Declaration

Fields/Variables

Constructor

Methods

Constructor

- A constructor is a method that creates an object.
- In Java, constructors are methods with the same name as their class used to create an instance of an object.
- Constructors are invoked using the new keyword.
- You can declare more than one constructor in a class declaration.
- You do not have to declare a constructor. In fact, Java will provide a default (blank) constructor for you.
- Example creating an object using Student constructor:

```java
Student stu = new Student();
```

Overloading Constructors

- Constructors assign initial values to instance variables of a class.
- Constructors inside a class are declared like methods.
- Overloading a constructor means having more than one constructor with the same name.
- However the number of arguments would be different, and/or the data types of the arguments would differ.

Constructor with Parameters

- A constructor with parameters is used when you want to initialize the private variables to values other than the default values.

```
public Student(String n, String ssn) {
   name = n;
   this.ssn = ssn;
}
```

- To instantiate a Student instance using the constructor with parameters, write:

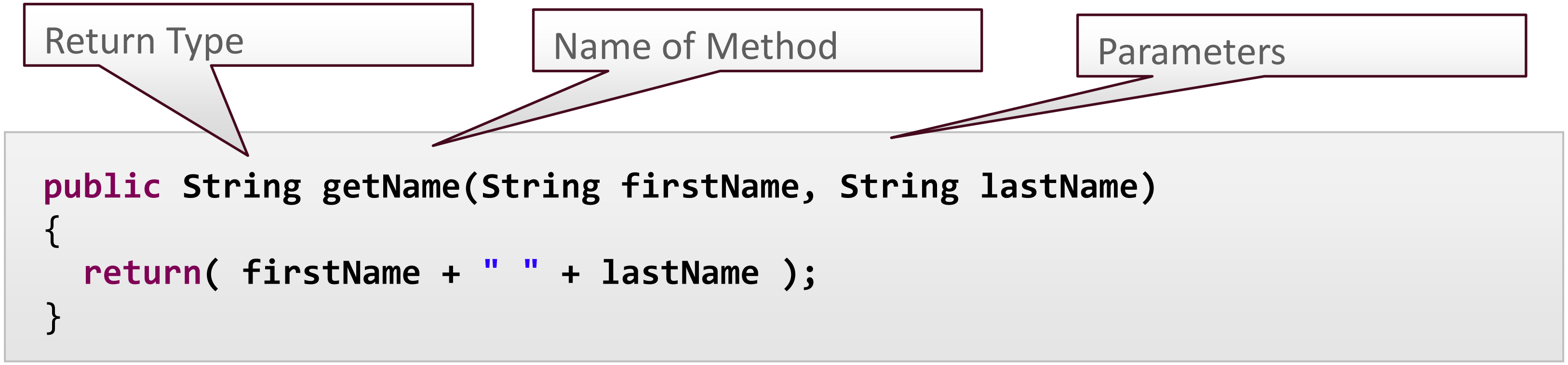
```
Student student1 = new Student("Zina", "3003456");
```

Components of a Method

- Method components include:
  - Return type:
    - This identifies what type of object if any will be returned when the method is invoked (called).
    - If nothing will be returned, the return type is declared as void.
- Method name: Used to make a call to the method.

Components of a Method

- Parameter(s):
  - The programmer may choose to include parameters depending on the purpose and function of the method.
  - Parameters can be of any primitive or type of object, but the parameter type used when calling the method must match the parameter type specified in the method definition.



Return Type

Name of Method

Parameters

```
public String getName(String firstName, String lastName)
{
  return( firstName + " " + lastName );
}
```

Class Methods

- Every class will have a set of methods associated with it which allow functionality for the class.
- Accessor method
  - Often called “getter” method.
  - Returns the value of a specific private variable.
- Mutator method
  - Often called “setter” method.
  - Changes or sets the value of a specific private variable.
- Functional method
  - Returns or performs some sort of functionality for the class.

Accessor Methods

- Accessor methods access and return the value of a specific private variable of the class.
- Non-void return type corresponds to the data type variable you are accessing.
- Include a return statement.
- Usually have no parameters

```
public String getName(){
  return name;
}
public int getStudentId(){
  return studentId;
}
```

Mutator Methods

- Mutator methods set or modify the value of a specified private variable of the class.
- Void return type.
- Parameter with a type that corresponds to the type of the variable being set.

```
public String setName(String name){
   this.name = name;
}
public int setStudentId(int id){
   studentId = id;
}
```

Functional Methods

- Functional methods perform a functionality for the class.
- Void or non-void return type.
- Parameters are optional and used depending on what is needed for the method's function.

Overloading Methods

- Like overloading constructors, overloading a method occurs when the type and/or number of parameters differ.
- Below is an example of a situation where a method would need to be overloaded.
- Create the Dog class, then create an instance of Dog in a Driver Class. Call (use) both bark() methods.

```
public class Dog{
  private int weight;
  private int loudness;
  private String BarkNoise;
  public void bark(String b){
    System.out.println(b);
  }
  public void bark(){
    System.out.println("Woof");
  }}
```

Main Method

- To run a Java program you must define a main method in a Driver Class.
- The main method is automatically called when the class is called.
- Example:

```
public class StudentTester
{
 public static void main(String args[])
 {
 }
}
```

Access Modifiers

- Access modifiers specify accessibility to changing variables, methods, and classes.
- There are four access modifiers in Java:

Access Modifier	Description
Public	Allows access from anywhere.
Protected	Allows access only inside the package containing the modifier.
Private	Only allows access from inside the same class.
Default (not specified/blank)	Allows access inside the class, subclass, or other classes of the same package as the modifier.

Superclass versus Subclass

- Classes can derive from or evolve out of parent classes, which means they contain the same methods and fields as their parents, but can be considered a more specialized form of their parent classes.
- The difference between a subclass and a superclass is as follows:

Superclass	Subclass
The more general class from which other classes derive their methods and data	The more specific class that derives or inherits from another class (the superclass)

Superclass versus Subclass

- Superclasses:
  - Contain methods and fields that are passed down to all of their subclasses.
- Subclasses:
  - Inherit methods and fields from their superclasses.
  - May define additional methods or fields that the superclass does not have.
  - May redefine (override) methods inherited from the superclass.

extends Keyword

- In Java, you have the choice of which class you want to inherit from by using the keyword extends.
- The keyword extends allows you to designate the superclass that has methods you want to inherit, or whose methods and data you want to extend.
- For example, to inherit methods from the Shape class, use extends when the Rectangle class is created.

```java
public class Rectangle extends Shape
{
//code
}
```

More About Inheritance

- Inheritance is a one-way street.
- Subclasses inherit from superclasses, but superclasses cannot access or inherit methods and data from their subclasses.
- This is just like how parents don't inherit genetic traits like hair color or eye color from their children.

Object: The Highest Superclass

- Every superclass implicitly extends the class Object.
- Object:
  - Is considered the highest and most general component of any hierarchy. It is the only class that does not have a superclass.
  - Contains very general methods which every class inherits.

Encapsulation

- Encapsulation is a fundamental concept in object oriented programming.

Encapsulation means to enclose something into a capsule or container, such as putting a letter in an envelope. In object-oriented programming, encapsulation encloses, or wraps, the internal workings of a Java instance/object.

How Encapsulation Works

- In object-oriented programming, encapsulation encloses, or wraps, the internal workings of a Java instance/object.
- Data variables or fields are hidden from the user of the object.
- Methods can provide access to the private data, but methods hide the implementation.
- Encapsulating your data prevents it from being modified by the user or other classes so that the data is not corrupted.

Terminology

Key terms used in this lesson included:

- Java Primitives
- Strings
- Logical and Relational Operators
- Conditional Statements
- Program Control
- Object Classes

Terminology

Key terms used in this lesson included:

- Constructor and Method Overloading
- Inheritance
- Encapsulation

Summary

In this lesson, you should have learned how to:

- Use Java Primitives
- Use of Strings
- Use Logical and Relational Operators
- Use Conditional Statements
- Use Program Control
- Understand Object Classes

Summary

In this lesson, you should have learned how to:

- Understand Constructor and Method Overloading
- Understand Inheritance
- Understand Encapsulation

ORACLE® ACADEMY