Java Language Overview

Key Terms and Concepts

- Java is an Object Oriented Programming (OOP) language.
- Programs in Java are organized around **classes**, with related classes being grouped together into **packages**.
- An **object** is created when we **instantiate** a class. We do this using the keyword "new", which calls the class **constructor**.
  - For example, new Cat("Ziggy") would call the constructor for the Cat class. Doing this *instantiates* a new Cat object.
  - **instantiate** - to make an instance of
- Classes can **extend** other classes, which means that they inherit all the functionality from those other classes (but they can also overwrite some of the functionality). In this case we call the extended class the *superclass* and the extending class the *subclass*.
  - For example, the Cat class may extend a Pet class. It may have some same functionality, e.g. weight, number of legs, age fields and accessors, but it may also have its own methods, e.g. purr.
- An **interface** is a set of **method signatures** (i.e., function prototypes). A class can **implement** an interface if it has implementations for all the methods indicated in the interface.
- The keyword this can be used in a method to refer to the object itself, and to provide access to the object's **data fields**.
- As an OOP language, much of Java programming boils down to calling object **methods**.

Access Modifiers

The **scope** of a class, method, or variable refers to the part of the program where the class/method/variable is *visible* and can be accessed. In Java, the scope of a class and the scope of each class attribute (method or data field) is specified using the following **access modifiers**:

public Can be accessed anywhere.

private Can be accessed only by objects of the class that contains them.

package-private Can be accessed by classes within the same package; also the default, if no access modifier is given.

protected Can be accessed only by subclasses of the class.

The scope is the first thing specified when a class or class attribute is being defined. In the example below, objects of type Foo can be created anywhere in the program. The Foo class methods are also public, which means these methods can be called on Foo objects anywhere in the program. However, The variable xBar is private, so the only place this variable can be accessed is inside of class Foo.

```
package fragle;
public class Foo {

  private int xBar = 42;

  public int getXBar() {
    return xBar;
  }

  public void addToXBar(int y) {
    xBar += y;
  }
}
```

Variables

Variables are identifiers in a program that hold values. Java employs several different kinds of variables, which vary in terms of their scope.

Instance variables Typically called **data fields** or just **fields**. Instance variables are part of a class's definition. They are created when an object is instantiated; each *object* is created with its own independent copy of the instance variables. All methods of an object can access the object's instance variables. If needed, instance variables may be referenced within an object through the this keyword; for example, this.x = x where the x on the left is the instance variable and the x on the right is a parameter or local variable.

Static variables Typically called **static fields**. Static variables are part of a class's definition. They are used to represent properties associated with a *class*. A single copy of each static variable is shared amongst all object instances of that class. Static variables can be used even if no class instances have been created; they are always referenced through the name of the class. For example, if class Zip has a static variable called numObjects, throughout the class the variable would be referenced as Zip.numObjects.

Local variables Typically just called **variables**. Local variables are variables defined inside of methods. They may only be referenced within the innermost set of curly braces that contains their declaration.

Parameters Sometimes called **arguments**. Parameters are declared in a method's signature. The values for parameters are passed to the method from its caller. A parameter may be referenced anywhere inside the method where it is defined, and its value only extends to the end of the specific method call.

A quick summary:

Variable	Scope	Lifetime
Instance	each object	as long as object is around
Static	any object of the class	always there
Local	innermost {. . . }	only within the {..}
Parameter	method	duration of method call

Practice Imagine a Student class. Which type of variable is best to use for each of the following pieces of information?

1. The variable holding the maximum allowed length for student last names.
2. The variable holding a student's current age
3. In a method called isOlderThan, which compares a student's current age to a provided target age, the variable that holds the provided age with which to compare the student's current age.
4. In the method isOlderThan, the boolean variable that holds the result of comparing the two ages.

Type of a variable

- Every variable needs to be **declared** before it can be used. This declaration specifies the type and scope of the variable.
- This **type** can be a built-in datatype like int or char, or it can be a class or interface.
- When assigning a value to a variable, the type of the value must match the type of the variable.

Syntax elements

Java program files

Java files use the extension .java. They consist of the following:

- *package* statement indicating which package the file belongs to
- *import* statements that load public elements from other packages
- class or interface definition

Import statements are used to avoid having to refer to classes by their fully qualified names.

Classes can be used without being imported by **fully qualifying** the class name. For example, graphics.Rectangle is how the Rectangle class would need to be referred to if the class was not imported.

Using import statements in your file allows you to use shorter forms to reference classes. Below are three possible import statements that could be used to import the Rectangle class.

```
import graphics.Rectangle;
import graphics.*;
import graphics.Rectangle.*;
```

- The first import line above allows us to refer to the Rectangle class directly in our code, e.g., new Rectangle.
- The second import line does the same for *all* classes within the graphics package.
- The third import line will make every *static* inner class and *static* method within the Rectangle class available directly.

As an example, if Rectangle had a static draw method, then with the first two imports we can refer to this method as Rectangle.draw(...), while with the last import we can do draw(...) instead.

Class definitions

Class definitions have the keyword class, possibly preceded by an *access modifier* and followed by the class name. In Java, it is conventional to *capitalize* class names. *Top-level classes must be contained in a file that has the same name as the class.*

The class name may be followed by extends ... and/or implements ... clauses, if the class is a subclass of another class or if it implements a certain interface. For example,

```
public class Circle extends AbstractShape implements Drawable {
    ...
}
```

The interior of a class definition may contain any of the following, in any order:

- field declarations
- zero or more constructor definitions
- method definitions
- inner class definitions

Interface definitions are similar to class definitions with the following exceptions:

- They cannot contain an extends part, unless it is to extend another interface.
- They cannot contain an implements part.
- They cannot have constructors.
- They have **method declarations** instead of method definitions.

Method declarations have no body; instead, a method declaration ends with a semicolon.

Abstract class definitions are similar to regular class definitions with the following exceptions.

- They cannot have constructors.
- They use the abstract modifier in their definitions.
- They can contain both method definitions and method declarations.

Methods that are simply declared *must* contain the abstract keyword modifier.

Field declarations

A field declaration specifies the visibility of the field, its data type, and possibly an initial value. Below are some examples:

- private String firstName;
  - Specifies a private field called firstname that is of type String. This field will likely be initialized in the constructor.
- static final int MAX_CAPACITY = 40;
  - Specifies what is effectively a *class constant*. MAX_CAPACITY is visible within the package (hence no access modifier in front). It is of type int and has been declared **final**, meaning its value cannot be change.
- public static String hostname;
  - Specifies a field called hostname that is visible everywhere. Using the static keyword modifier indicates that hostname is a *class* variable, with one copy of the variable being shared by all class objects.

Method definitions

Method definitions begin with *modifiers* followed by the method's *return type*, the method name, and a parenthesized list of parameter declarations. The body of the method then follows, enclosed in curly braces. Below are some examples:

```
public void setFirstName(String newFirstName) {
    ...
}

public String getFirstName() {
    ...
}

private static boolean isValidName(firstName, lastName) {
    ...
}
```

Note that if your program has a **main method**, this method has a specific required signature. For example:

```
public static void main(String[] args) {
    System.out.println("Hello␣world!");
}
```

Constructors

Constructors are similar to method definitions. The name of a class also functions as the name for the class constructor. *Contructors cannot be static.* They also do not explicitly return a value from within their body. Instead, the newly created object is automatically returned.

A class can have multiple constructors as long as the constructors have differences in their parameter lists. When a class has multiple constructors, it is customary for all constructors to eventually call the same constructor, the one with the most complete set of arguments.

A class implemented without a constructor will be given the following *default constructor*: `public void ClassName() {}`.

Example:

```
public Person(String firstName, String lastName, int age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
}

// This constructor simply calls the longer constructor.
public Person(String firstName, String lastName) {
    this(firstName, lastName, 0);
}
```

Constructors are called when the keyword `new` is used to create a class instance. For example, `Person p = new Person("Peter", "Doe", 26);` will automatically call the `Person` class constructor.

Control structures

Java contains many of the standard control elements you may have seen in other languages to control the flow of execution:

- conditionals
- while loops
- for loops
- switch statements

Conditionals Java conditionals follow a standard if−then−else pattern:

```
if (thisIsTrue) {
    ... true case code
}

if (thisIsTrue) {
    ... true case code
} else {
    ... false case code
}

if (thisIsTrue) {
    ... true case code
} else if (thatIsTrue) {
    ... case where thatIsTrue is true but thisIsTrue is false
} else {
    ... both false
}
```

There is also the **ternary operator**, which *returns a value*. For example,

```
int x = (y > 5) ? 5 : y;
```

This can be translated as "if y is greater than 5 then return 5; otherwise, return the value of y".

While loops Java implements the familiar while loop that executes a block of code as long as a specific condition is true. For example,

```
while (account.hasMoney()) {
    account.withdraw();
}
```

This loop will continue executing the account.withdraw() method as long as account.hasMoney() is true.

For loops Java provides two kinds of for loops. The standard *counter loop* and a *foreach loop* for iterating over a collection.

```java
for (int i = 0; i < allNames.length; i++) {
    System.out.println(allNames[i]);
}

for (String name : names) {
    System.out.println(name);
}
```

The second for loop will iterate over all the items in names using the variable name as its loop variable. This kind of for loop is preferred if you simply need to loop over the elements of a collection and you have no need for the loop index i. And as an extra benefit, it works for all sorts of *iterables* and not just for arrays. Any object whose class implements the Iterable interface[1] can be used in this so-called **for-each loop**.

Switch statements A switch statement compares a value against a series of constant values presented in case expressions, and executes the corresponding statements. It will fall through to the next case unless you remember to use break.

```java
switch (person.getType()) {
    case "faculty":
        doSomething();
        break;
    case "student":
        doSomethingElse();
        break;
    default:
        // We don't do anything otherwise
}
```

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Iterable.html