CITS5501 Java revision

Arran Stewart

2023-02

Introduction

Java is an object-oriented programming language developed by Sun Microsystems and now owned by Oracle Corporation. It is designed to be platform-independent, meaning that Java code can be written once and run on any system that has a Java Virtual Machine (JVM) installed, regardless of the underlying hardware and operating system. Its syntax is similar to that of C++.

Java code is normally compiled into *bytecode*. This is a low-level representation of code, similar to machine code, which is designed to be executed by a virtual machine rather than directly by the computer's hardware. The original source code is translated into a series of bytes representing machine-readable instructions.

Java is widely used for building enterprise applications, and the JVM is used in the development of Android applications.

Useful texts. The text *Objects First with Java: A Practical Introduction Using BlueJ* (Barnes and Kolling, 2016) provides a good introduction to the object-oriented paradigm for beginning programmers using the Java language, and is available through the UWA library. For more experienced programmers, *Java in a Nutshell: A Desktop Quick Reference* (Evans and Flanagan, 2019) may be useful. The definitive reference on Java is the *Java Language Specification* (Gosling et al, 2022).

For students already familiar with the Java language, we provide a brief refresher of some relevant concepts here. It is *not* a complete reference; you should refer to the texts mentioned above for that.

Java types and expressions

Java is a reasonably simple language (if somewhat verbose), and is *statically typed*. This means that the types of variables are known at compile time. If a variable is declared as holding a `boolean`, then it is impossible to assign to that variable a value of the wrong type; the program simply won't compile. This is in contrast to *dynamically typed* languages like Python, where the type of a variable is only known at runtime, and can change – it's perfectly possible for one variable to be assigned a `bool`, then an `int`, then a string. In this unit, you are expected to understand the difference between statically and dynamically typed languages.

Java types are divided into two categories: primitive types and reference types. The primitive types (discussed in the next section) are the `boolean` type and the numeric types. The most important reference types, for our purposes, are class types, interface types, and array types.

Primitive types

Java has eight primitive types, as follows:

Type	Description
boolean	Holds a 'true' or 'false' value
char	Holds one Unicode character of 16 bits size. They can be written as character literals, which have single quotes around them (e.g. `char c = 'A';`).
byte	Holds a signed integer of size 8 bits (ranging from -128 to 127).
short	Holds a signed integer of size 16 bits (ranging from -2^{15} to $2^{15}-1$).
int	Holds a signed integer of size 32 bits (ranging from -2^{31} to $2^{31}-1$).
long	Holds a signed integer of size 64 bits (ranging from -2^{63} to $2^{63}-1$).
float	Holds an IEEE 754 floating point value of 32 bits size.
double	Holds an IEEE 754 floating point value of 64 bits size.

Java performs *"widening"* conversions between primitive types automatically. For instance, an `int` can represent all the values a `short` can hold and more, so it is considered a "wider" type. It's therefore possible to assign a `short` value to an `int` variable, and Java will perform the appropriate conversion automatically:

```
1  short s = 99;
2  int i = s;    // compiles with no issues
```

Note that Java does not automatically convert between booleans and numeric types, as some other languages do. (For instance, in C, C++, and Python, an integer value of 0 is considered equivalent to boolean "false".)

Basic Java operators

Java *expressions* consist of variables and literals joined by operators. Java's operators are similar to those found in C++ (and for that matter, Python and many other modern programming languages).

They include arithmetic operators (`+`, `-`, `*`, `/`, `%`), logical operators (`&&`, `||`, `!`), and relational operators (`<`, `<=`, `>`, `>=`, `==`, `!=`).

One difference to Python is that Java uses symbols for its logical operators, whereas Python uses words ("`and`", "`or`", and "`not`"). Another is that when the division operator is used with two integers, it performs *integer division*: it returns another integral value, not a floating point type. Thus, `10 / 3 == 3`.

Be careful when using any of these operands with reference types. When the `==` operator is used with reference types, it tests whether the operands refer to the same object or array – it does *not* test the "equality" of two distinct objects or arrays.[1]

Java includes the somewhat unusual feature of *non-short-circuiting* logical operators. The `&` operator implements non-short-circuiting "and", and the `|` operator implements non-short-circuiting "or". When these operators are used, *both* operands get evaluated, even if they don't need to be. In the following example:

```
1  bool b = false & someObject.someMethod();
```

the method `someMethod` gets invoked, even though we know the value of `b` *must* be false.

Special operators

Some other important Java operators are as follows.

Object member access (`.`) Given some object, the dot (`.`) operator lets you access its fields and methods. E.g. `"wibble".length()` invokes the `length()` method of the string `"wibble"`.

Array element access (`[]`) Given an array, the `[ ]` operator lets you refer to a specific element of an array (much the same as most other languages). Thus if `arr` is an array, then `arr[0]` refers to its first element.

Method invovation (`()`) Given some method, putting parentheses after it (possibly with arguments between them) will *invoke* that method.

Object creation (`new`) In Java, objects and arrays are normally created with the `new` operator. So, for instance

```
1  Queue myQueue = new Queue();
```

[1] In other words, when used with reference types, the Java expression `obj1 == obj2` is *not* equivalent to the Python expression `obj1 == obj2`; it is closer in semantics to the Python expression `obj1 is obj2`.

will construct a new `Queue` object (used to represent a "first-in, first-out" queue).

Strings

Strings in Java are *not* a primitive type, but a reference type. `String` in Java is a class, and follows the Java convention of having a "PascalCase" name (with an initial capital letter).

Although `String` is a class, strings are treated specially in Java in several ways. One is that they can be constructed using *string literals*, which are written using double quotes:

```
1  String s = "abracadabra";
```

Strings are also *immutable* – a string value cannot be altered once constructed. (So in this respect, they are different to an array of `char`s, where once the array is created, you can, for instance, change the first character to something else.) For a mutable type which allows you to construct strings, see the `StringBuilder` class.

Because they're not primitive types, strings cannot be compared using the `==` operator. When used on objects, `==` tells you if the left and right operands *are the same object*, whereas we typically want to know if they hold equivalent strings. Instead, you need to use the `.equals()` method to check for string equality:

```
1  String s1 = "cat";
2  String s2 = "dog";
3  String s3 = "cat";
4  System.out.println( s1.equals(s2) ); // prints "false"
5  System.out.println( s1.equals(s3) ); // prints "true"
```

Strings can be concatenated using the `+` operator:

```
1  String s = "abraca" + "dabra";
```

Strings are considered "wider" than any of the primitive types, so if you concatenate a string and a number, the number is automatically converted to a string:

```
1  String s = "agent " + 99
2  System.out.println(s); // will print "agent 99"
```

Objects are converted to strings by calling their `toString()` method.

Classes and objects

In Java, an object is a set of *fields* (which hold data) and *methods* (which operate on that data). A class is a blueprint or template for creating objects.

A class to represent a point on a Cartesian plane might look like this:

```
/** Represents a Cartesian (x,y) point */
public class Point {
  // coordinates of the point
  private double x, y;
  // constructor
  public Point(double x, double y) {
    this.x = x; this.y = y;
  }
  // a method
  public double distanceFromOrigin() {
    return Math.sqrt(x*x + y*y);
  }
}
```

Once a class is defined, we can use the `new` operator to instantiate objects of that class – e.g. `Point p = new Point(3.0, 2.0)` would create a new `Point` object.

The `null` keyword is a special type of literal value in Java that represents *nothing* – the absence of any object or reference. The `null` value can be assigned to variables of any reference type:

```
String s = null;
Point p = null;
```

Classes are the most important of the *reference types* supported by Java. A class in Java may inherit from another class, using the keyword "`extends`":

```
import java.awt.Color;

/** a Cartesian point with colour */
public class ColouredPoint extends Point {
  // colour of the point
  private Color color;
  // constructor
  public ColouredPoint(double x, double y, Color color) {
    super(x,y);
    this.color = color;
  }
  // ... other methods ...
}
```

Inheritance allows programmers to reuse properties and behaviours from existing classes, and build more specialized classes that inherit the properties and behaviours of a parent class (it's *direct superclass*).

If a class doesn't explicitly inherit from some other class using the "`extends`" keyword, then it automatically inherits from a special class called `Object`. Thus, every Java class has `Object` as a (direct or indirect) superclass.

Arrays

Arrays are a special kind of object which can hold zero or more values of some type in contiguous *elements* in memory. We will not need to directly use arrays much in this unit, and will instead use classes from the Java Collections Framework, such as `Set`, `List`, and `TreeMap`.

Interfaces

In Java, classes can only inherit from *one* other class, which is a significant restriction in an object-oriented language. To ameliorate this, Java introduced interfaces. Like a class, an interface lets a programmer define a new reference type. However, it is an abstract type, containing no implementations for its methods; all it defines is the *interface* for the type (hence the name).

For instance, an interface `Shape` might be defined as follows:

```
1  public interface Shape {
2    double getArea();
3    double getPerimeter();
4  }
```

A Java class (or interface) can declare that it implements any number of interfaces, using the "`implements`" keyword. For instance:

```
1  public class Rectangle implements Shape {
2    double width, height;
3    double getArea() { return width * height; }
4    double getPerimeter() { return width * 2 + height * 2; }
5    // .. constructor and other methods omitted
6  }
```

Reference types versus primitive types

Reference types differ from primitive types in several important ways:

- **Defining new types**. It's not possible to define new primitive types in Java. The eight predefined types are the only ones possible. In contrast, programmers can define as many new reference types (for instance, new classes or interfaces) as they like.

- **Primitive types represent exactly one value; reference types may represent zero, one or more**. A Java `int`, for example, represents a single 32-bit signed integer. But reference types are *aggregate* types that can represent zero or more primitive values or objects. An array type like `char[]` represents a sequence of primitive types; an array type like `String[]` represents a sequence of objects (all of type `String`).

  For classes, the values represented by the type are given names and are called fields; our `Point` class above holds two `double` values and gives them the names `x` and `y`.

- **Memory used**. When you declare a variable of a primitive type, it always occupies some fixed amount of memory that will be between one and eight bytes, and will vary depending on the type. (A `boolean` occupies exactly one byte, a `long` eight.) Reference types, on the other hand, often require more memory than that. The memory is dynamically allocated when the object is created, and automatically "garbage collected" when the object is no longer needed (e.g. because it goes out of scope).

- **How values are passed**. When a value of some primitive type is passed to a method, the method receives a *copy* of the value. (This is called "passing by value".) This means any changes made by a method to a parameter are *not* seen by the calling method, since the called method operates only on a copy of the original. In the following code –

```
1  public static void addOne(int i) {
2    i += 1;
3  }
4
5  public static void main(String[] args) {
6    int j = 5;
7    addOne(j);
8  }
```

  calling `addOne()` has no effect on the value of `j`.

  Reference types, however, are passed "by reference"; this means that a called method potentially has full access to the original object, and can mutate it just as the calling method could. In the following code:

```
1  public static void setFirstEl(int[] arr) {
2    arr[0] = 42;
3  }
4
5  public static void main(String[] args) {
6    int[] myarr = { 0 };
7    setFirstEl(myarr);
8  }
```

calling `setFirstEl()` *does* result in a change to `myarr`.

Sometimes it's useful to be able to treat Java's primitive types as if they were more like reference types. For instance, the Collections Framework provides an `ArrayList` type which we can use to represent lists of things; an `ArrayList<Color>` represents a list of colours. But Java's collections can only hold *reference* types[2], not primitive types: trying to declare an `ArrayList<int>`, for instance, will result in a compilation error. For each primitive type, Java therefore defines a utility class which "wraps" that type. You can see a list of these "wrapper" classes here. For convenience, Java will automatically convert between a primitive type and its corresponding wrapper type; thus, one can write

```
1  Integer i = 0;
```

instead of

```
1  Integer i = new Integer(0);
```

Other reference types

There are a few other reference types we have not yet mentioned. Java's enums are similar to enums in Python or C, and allow a programmer to create a new type which consists of a set of predefined constants.

Java annotations give programmers a way of attaching metadata to parts of a Java program. An annotation could be used, for instance, to specify the author who created a particular Java class. In CITS5501, we will make use of Java annotations provided by the JUnit testing framework. One such annotation, `@Test`, allows us to mark particular methods of a class as **JUnit tests** which should be executed by a test runner:

```
1  import static org.junit.jupiter.api.Assertions.assertEquals;
2  import org.junit.jupiter.api.Test;
3
4  class ArithmeticWorks {
5    @Test
6    void additionWorks() {
7      assertEquals(4, 3 + 1);
8    }
9  }
```

Other useful Java features

Two features of Java you will likely find useful in the unit are *generics* and *lambda expressions*. For a full discussion of these, you should refer to a Java textbook. But in

[2]This is a limitation of Java's generic types.

brief:

- Java generic types are classes and interfaces that are *parameterized over types*. Similar to the way a mathematical function, when supplied with appropriate parameters, "spits out" a result, a generic type, when supplied with appropriate parameters, "spits out" a new type.

  `HashSet<T>` from the Java Collections Framework is an example of a generic type. On its own, it doesn't represent a type that you can directly instantiate as an object in your program. But if supplied with an appropriate *type parameter*, it does. Thus `HashSet<String>` could be instantiated with code like the following:

```
1  HashSet<String> mySet = new HashSet<>();
```

  (Java lets us omit `<String>` the second time it would appear, as the compiler is able to infer the missing type from context.)

- A Java lambda expression is a concise way of representing an anonymous function.[3] A simple lambda expression which takes two parameters and returns a result would take the form:

  `(`*`param1`*`,` *`param2`*`) -> `*`expression`*

  For instance,

```
1  (x, y) -> x+y
```

  is a lambda expression which adds its arguments together. Types *can* be included before the parameters, but need not be. The expression can be stored in a variable or passed to a method.

  Lambda expressions were introduced in Java 8, and are roughly equivalent to a class with a single method (which we might call `apply`) – something like the following (if we specialize the expression to `Integers`), but without needing a name like `MyLambda`:

```
1  class MyLambda {
2    public static Integer apply(Integer x, Integer y) {
3      return x + y;
4    }
5  }
```

  But they are not *exactly* equivalent, and using lambda expressions generates bytecode that cannot be interpreted by pre–Java 8 JVMs.

[3] Java lambda expressions are equivalent to lambda expressions in Python, C++, JavaScript, and Rust. The first language to make use of lambda expressions was Lisp, in 1958.

Java programs

You should refer to a Java textbook for full details of how Java programs are constructed and compiled.

In brief, however, Java programs consist of one or more Java files, which by convention have a `.java` extension. A Java compiler (normally, `javac`) compiles each file into one or more bytecode files, which have a `.class` extension.

A Java program consists of some set of interacting class definitions, where at least one class has a `main()` method declared with the following signature:

```
1  public static void main(String[] args);
```

There can be multiple classes with `main()` methods, providing multiple ways of invoking the program.

References

- Barnes, David, and Michael Kolling. *Objects First with Java: A Practical Introduction Using BlueJ*. 6th edition. Boston: Pearson, 2016.
- Evans, Benjamin, and David Flanagan. *Java in a Nutshell: A Desktop Quick Reference*. 7th edition. Beijing, China; Boston, Massachusetts: O'Reilly Media, 2019.
- Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java Language Specification*. Oracle, August 31, 2022. Available at https://docs.oracle.com/javase/specs/jls/se19/html/index.html.