Chapter 3 Using Classes and Objects

p.250-313

3.1 Creating Objects

Like primitives, object variables must be **declared**.

The **class** of the object can be thought of as its **type**.

When a class variable is declared, the variable holds a reference (or “points”) to the object’s address in memory.

Initialization

As with primitive variables, object variables can be declared but not initialized, ex: `String name;` . Ensure objects are initialized before they are used.

An uninitialized object variable is equal to `null` , which is a Java reserved word meaning “no value”.

Object variables can also be assigned to `null` later, to clear their value.

Creating a new instance of an Object is called **instantiation:**

```
name = new String("Shivaun");
```

the `new` keyword tells java that an object is being instantiated.

`String("Shivaun")` is a special method inside the `String` class called a **constructor**. Constructors are methods that set up new instances of a class.

Accessing Object Methods

Methods are functions that exist inside a class, defining the class’s behavior.

We can use the dot operator to access the methods of an instantiated object.

For example: `name.length()` would print the length of `name` in characters (in this case, 7).

Note that even though the `length` method has no arguments, we still must include the empty parentheses for proper syntax when invoking a method.

Aliases

Because object variables hold a **reference** to their location in the Java Virtual Machine’s memory, they work subtly differently than primitives.

When one primitive variable is assigned to another, it simply pass its value to the new variable. Eg:

int a = 5;

int b = 7;

a = b;

In this example, even though a and b both equal 7, they still point to two different registers in memory.

Unlike primitives, **multiple reference variables can point to the same object**. This means that modifying any variable pointing to that object will likewise affect all other such variables.

String name1 = "bob";

String name2 = "fred";

name1 = name2;

In this example, name1 and name2 do *not* point to two different registers, each containing “bob”. Rather, they both point to the same register in memory, which contains the String object.

We call the concept of multiple variables pointing to the same object **aliases.**

Because all interaction with objects happens through a reference variable, when all references to an object are lost, it becomes impossible to use that object in the program. We call such objects **garbage**.

Java performs **automatic garbage collection**, meaning that behind the scenes it periodically frees up the memory used by garbage to be reassigned to something useful.

3.2 The `String` Class

As with most classes, `String` has a number of **methods**. Each method has a return type and 0 or more arguments.

A unique property of the `String` class is that it can be instantiated using a literal (double quotation marks) without explicitly using the `new` operator and constructor call:

```
String name = "Shivaun"
```

There is also a special syntax for text blocks: use `"""` before and after the text block.

The `String` object is **immutable**, meaning its value cannot be changed once initialized. String modifications always result in a new `String` object.

Constructor:

- `String(String str)` - Creates new string object with same characters as str

Character Operations:

- `charAt(int index)` - Gets character at specified index
- `length()` - Returns number of characters in string

String Comparison:

- `compareTo(String str)` - Compares strings character-by-character lexically, returning a negative value if the first string is less, a positive value if it is greater, or `0` if they are equal.
- `equals(String str)` - Checks if strings match (case-sensitive)
- `equalsIgnoreCase(String str)` - Checks if strings match (case-insensitive)

String Manipulation:

- `concat(String str)` - Joins two strings together
- `replace(char oldChar, char newChar)` - Replaces all instances of a character
- `substring(int offset, int endIndex)` - Gets portion of string from offset to endIndex-1
- `toLowerCase()` - Converts string to lowercase
- `toUpperCase()` - Converts string to uppercase

3.3 Packages

To deal with complexity, language design has added many structures in higher level languages to make it **modular.**

- Statements
- Functions/methods
- Classes
- Packages
- Modules/Components/Subsystems

A **class library** is a set of classes that supports the development of programs. Java comes with a class library called the *Java standard class library*.

Class libraries can be obtained through 3rd party sources like Github.

Many of the features we think of as part of Java itself, such as the `String` class, are actually part of the Java standard class library.

The standard class library is broken into broad categories called **API**s (application programming interfaces)

As a whole, the Java standard class library is also sometimes called the **Java API** what is the difference between an API and a package

The classes of the Java standard class library are grouped into **packages.** An API may contain several packages.

A few packages in the Java API:

Package	Description
java.awt	Graphics and GUI components (Abstract Windowing Toolkit)
java.beans	Reusable software components for applications
java.io	Input/output operations
java.lang	Core language support (auto-imported)
java.math	High-precision calculations
java.net	Network communications
java.rmi	Remote Method Invocation for distributed computing
java.security	Security and access control
java.sql	Database connectivity (SQL - Structured Query Language)
java.text	Text formatting utilities
java.util	General utility classes

javafx.scene.shape	Geometric shapes for graphics
javafx.scene.control	GUI controls (buttons, sliders, etc.)

The **Java Platform Module System** partitions Java libraries into commonly used subsystems.

What is a **module**?

- a collection of packages
- optionally, resource files and native libraries
- a list of packages in the module.
- a list of modules the module depends on

Modules allow large systems to designed with reusable components, increasing scalability.

Modules are defined by `module-info.java` in the base directory.

`java.lang`

The `String` and `System` are part of a special package called `java.lang`. It is the only package that is implicitly imported into every Java document, making it always accessible. `java.lang` contains many useful classes, including:

- Primitive Wrapper Classes: `Integer`, `Double`, `Boolean`, etc.
- Utility Classes: `Math`, `String`, `Object`, `System`, `Thread`, etc.
- Exception Handling Classes: `Exception`, `RuntimeException`, `Error`

The `import` Declaration

All packages other than `java.lang` must be **imported** to be made accessible in a Java file.

There are two ways to do this:

- Fully qualify the reference inline. Eg: `java.util.Scanner scanner = new java.util.Scanner(System.in);` This method is verbose and harder to read, so we prefer the second:
- Include `import` declaration(s) before declaring the Class. Eg: `import java.util.Scanner;`

In a file containing this declaration, we can simplify the line above to `Scanner scanner = new Scanner(System.in);`

The `*` character can be included in an import declaration to import all classes within a package. Eg:

```
import java.util.*;
```

This is useful if two or more classes from the package will be used, but if you are only using one, it is better to explicitly import it.

Packages can be defined easily:

As the first line of the source code, make a package statement:

```
package mypackagename;
```

```
package ca.bcit.infosys.servletutils;
```

The source code file must be in a directory (directories) corresponding to the package name (with periods replaced with directory separator)

3.4 The `Random` Class

The `Random` class from `java.util` generates **pseudorandom** numbers.

Pseudorandom means *seemingly random*, but not truly random, because computers have no means to actually pick a number randomly.

`Random` uses a **seed value** to generate a random number. The same seed will always generate the same values.

Some methods of `Random`

`Random()` - Constructor: creates a new pseudorandom number generator.

`nextFloat()` - Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

`nextInt()` - Returns a random number that ranges over all possible int values (positive and negative).

`nextInt(int num)` - Returns a random number in the range 0 to num-1.

3.5 The `Math` Class

The `Math` class from `java.lang` provides a large number of basic math functions.

All the methods in the `Math` class are **static methods** (also called class methods). This means they can be invoked from the class directly (through the class name) without needing an instance.

Math Class Methods

Basic Math Operations

`static int abs(int num)` - Returns absolute value

`static double pow(double num, double power)` - Returns num to specified power

`static double sqrt(double num)` - Returns square root (num must be positive)

Trigonometric Functions

`static double cos(double angle)` - Returns cosine. Functions also exist for sin and tan.

`static double acos(double num)` - Returns arc cosine. Functions also exist for arc sine or tan.

Rounding

`static double ceil(double num)` - Rounds up to nearest whole number

`static double floor(double num)` - Rounds down to nearest whole number

Random Number Generation

`static double random()` - Returns random number between 0.0 (inclusive) and 1.0 (exclusive)

3.6 Formatting Output

The `NumberFormat` and `DecimalFormat` classes in the `java.text` package are used to format information.

`NumberFormat`

The `NumberFormat` class is not instantiated with new. Rather, it uses a specialized static method that provides a specialized `NumberFormat` instance, ex:

`static NumberFormat getCurrencyInstance()` - Returns a `NumberFormat` that formats numbers as currency.

`static NumberFormat getPercentInstance()` - Returns a `NumberFormat` that formats numbers as percentages.

Once you have an instance of `NumberFormat`, it can be used via the `format` method:

`String format(double number)` - Returns a string containing the specified number formatted according to this object's pattern.

`DecimalFormat`

Unlike `NumberFormat` , the `DecimalFormat` class *is* instantiated the traditional way with the `new` operator. It’ss constructor takes a string that represents the pattern that will guide the formatting process.

`DecimalFormat(String pattern)` - Constructor that creates a new DecimalFormat object with the specified pattern.

`void applyPattern(String pattern)` - Applies the specified pattern to this DecimalFormat object.

`String format(double number)` - Returns a string containing the specified number formatted according to the current pattern.

The **patterns** used by `DecimalFormat` have their own syntax:

1. **Digit Representation:**
    - `#`: Optional digit; displays a digit if present, otherwise skips it.
    - `0`: Mandatory digit; displays a digit or a zero if none is present.
2. **Decimal Separator:**
    - `.`: Specifies the decimal point.
3. **Grouping Separator:**
    - `,`: Groups digits, typically used for thousands (e.g., `#,###`).
4. **Prefix/Suffix:**
    - Add text or symbols before/after the pattern (e.g., `¤` for currency, `%` for percentages).
5. **Exponent:**
    - `E`: Separates the mantissa and exponent for scientific notation (e.g., `0.###E0`).
6. **Escape Literal Characters:**
    - `'`: Encloses literal text (e.g., `'USD '0.00`).

Examples:

- `#,###.##` → Displays numbers with commas for grouping and up to 2 decimal places.
    - Input: `12345.678` → Output: `12,345.68`
- `000.00` → Pads with leading zeros and always shows 2 decimal places.
    - Input: `5.2` → Output: `005.20`
- `$#,###.00` → Adds a dollar sign and formats with commas and 2 decimals.
    - Input: `1234.5` → Output: `$1,234.50`
- `0.###E0` → Formats in scientific notation with up to 3 decimal places.
    - Input: `12345` → Output: `1.235E4`

The `printf` Method

The `printf` method in the `System` class prints a formatted string containing data values.

`printf` is a legacy method that exists to make migrating software from C to Java easier (C has a similar method). It isn't an elegant solution to string formatting and is not often used.

Ex.

```
System.out.printf("ID: %5d\\tName: %s", id, name);
```

might return an output like…

```
ID: 24036 Name: Larry Flagelhopper
```

3.7 Enumerated Types

An **enumerated type** (enum) is a special data type that defines a fixed set of constant values. It is used when you know all possible values at compile time, such as days of the week, months, or card suits.

Basic syntax for declaring an enum:

```
enum Season {winter, spring, summer, fall}
```

Example usage:

Season currentSeason = Season.winter;

if (currentSeason == Season.winter) {

    System.out.println("Do you want to build a snowman?");

}

Key characteristics of enums:

- Enums are **type-safe**, meaning an enum variable can only hold one of the enum's predefined values, and nothing else.
- Each value in an enum is stored as an integer called its **ordinal value**. The first value in the type is 0, the second is 1, and so on. These numbers are only used internally - you can't assign an enum an integer value.
- enums are a special type of class, so enum variables are object variables.

Enums have a few built-in methods:

- `ordinal()` - Returns the numeric value (position) of the enum constant
- `name()` - Returns the string name of the enum constant

3.8 Wrapper Classes

A wrapper class represents a particular primitive type. For instance, the `Integer` class represents a simple integer value.

Eg:

An object created from the Integer class stores a single `int` value.

```
Integer ageObj = 40;
```

just a note here: the nextbook uses the example: `Integer ageObj = new Integer(40);` but you will get a depreciation error if you try to run this code. Using constructor methods with wrapper classes was made unnecessary in Java 9. You can simply assign them a primitive literal.

This automatic conversion to/from a primitive value to a corresponding wrapper object is called **autoboxing**, and is now the standard way of instantiating Wrapper classes.

Autoboxing can be used to turn a wrapper back into a primitive (unboxing):

```
int age = ageObject;
```

A wrapper variable can be used the same as we use primitive variables of the same type:

```
Integer i = 40;

 i += 5;

 System.out.print(i);

 //output: 45
```

On the surface, it may seem like wrappers do the same thing as their primitive counterparts, but because they are objects, variables hold them as references rather than values. **Wrapper classes allows primitive values to be managed as objects.**

All the Wrapper classes in Java are listed below:

Primitive Type	Wrapper Class
byte	Byte
short	Short

int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean
void	Void

Note that there is even a wrapper class that represents the type void . However, unlike the other wrapper classes, the Void class cannot be instantiated. It simply represents the concept of a void reference

The Wrapper classes also provide useful methods related to their primitive type. For example, the `Integer` class provides the following methods:

`Integer(int value)` Constructor: creates a new Integer object storing the specified value.

`byte byteValue()` `double doubleValue()` `float floatValue()` `int intValue()` `long longValue()` Return the value of this Integer as the corresponding primitive type.

`static int parseInt(String str)` Returns the int corresponding to the value stored in the specified string.

`static String toBinaryString(int num)` `static String toHexString(int num)` `static String toOctalString(int num)` Returns a string representation of the specified integer value in the corresponding base.

Wrapper classes also provide useful static constants:

- The `Integer` class includes:
  - `MIN_VALUE`: Smallest possible int value
  - `MAX_VALUE`: Largest possible int value

Similar constants exist for other wrapper classes to define their type limits.