**Section 2: Java Data Types**

# Variable declaration and initialization (8 min)

In the previous section we covered the topic of fields and saw various examples of how to assign a return value to a variable and access a method parameter.

You learnt about variable scope and why it is important.

Now you are going to learn about variable declaration and initialization, variable default values and how that differs between local, instance and class variables.

Before a variable can be used it must be declared, this means its type is specified and then initialised: the variable is given a value. The initial value of a variable may be the default value specified by the Java language, this is determined by the type of the variable and depends on weather or nor the variable is a local variable, an instance variable or a class variable.

## How to declare a variable

A variable is declared by specifying its type followed by a legal identifier. Remember back to the discussion on fields in the previous section. Fields are variables defined at the class level and are also referred to as class or instance variables. A variable’s type can be either an object type or a primitive.

## How to Initialise a Variable

Initialising a variable means giving it a value. This can be done by explicitly setting a value or, if it is a class variable, by letting the language give it a default value.

## Class and Instance Variables

Class and instance (field) variable are the only variable class that is given an initial (default) value by the language as soon as it is declared. The value it is given is determined by its type: which will be *0* for a primitive, *false* for a boolean and null for an object reference.

The table below shows the default values for all possible declared types.

|  |  |
| --- | --- |
| **Declared Type** | **Initialization Value** |
| byte, short, int, long | 0 |
| char | \u0000 |
| float, double | 0.0 |
| boolean | false |
| Object References (e.g. BankAccount) | null |

Ensure that you learn these default values as you may need this knowledge to answer questions in the exam.

## Local Variables

In contrast to instance and class variables, local variables are declared only within a method and DO NOT have an initial value assigned. A value must be given to a local variable before it is used. The compiler will not let code read from an uninitialized variable and failing to assign a value will result in a compile time error.

## Review of Variable Scope

In the previous subsection: **Java Fundamentals** we discussed variable scope in some detail. Now that we know a little bit more about variables lets recap what we know about the rules of scope.

Class variables have a scope that starts when they are declared and lasts until the application terminates. Class variables are declared with the keyword *static* and are declared as soon as the class is loaded by the JVM which, in most cases, will be when the application is launched. Class variables have the longest lived scope.

Instance variables are in scope from the moment of declaration until they are garbage collected. We talk about garbage collection later on in this section.

Declaration happens when an instance of the class is created via instantiation. We talk about this process later on as well.

Local variables have the shortest lived scope. It starts when the variable is declared and finishes at the end of the code block.

## Quiz

Which of the following are legal ways to declare a variable?

1. String cat, mouse;
2. i int;
3. char a, b, c, d, e, f;
4. BankAccount ba;
5. String name, char letter;

Answer: 1, 2, 4

Which of the following ways are legal ways to declare and initialise variables?

1. int a, b, c; a, b, c = 100;
2. String bat, ball, stick = "wood";
3. byte yum = 127, byte no = 0;
4. char letter = 'A'; int code = 555;
5. float f; g = 808f;

Answer: 2, 4

Examine the following code snippet and identify the lines that are illegal:

1: public void DeclareLocalVarables(){  
2: int x = 100;  
3: int z, y = 200;  
4: int result = x = 5;  
5: int score = (y = 5) + y ;  
6: int tally = x + y + z ;  
7: }

1. line 3
2. line 4
3. line 5
4. line 6
5. none of the above

Answer: 4

Look at the class in the code snippet below. What is the final value of total?

public class DefaultValueQuiz {  
 int points = 10;  
 int bonus;  
 public void calculate(){  
 int total = 100;  
 total = points + bonus;  
 }  
}

1. 0
2. 10
3. 100
4. 110
5. The code does not compile

Answer: 2

# Understand object references and primitive variables (8 min)

## Primitive Data Types

We have talked about a variable’s default value; now let’s talk about a variables maximum and minimum value. The idea that a variable can have a range of values only applies to primitive types.

A primitive type, in contrast to an object reference, is the most basic data type in the Java language. There are eight of these types: *boolean*, *byte*, *char*, *short*, *int*, *long*, *float* and *double* and they can be divided in three broad ways: *numeric*, *textual* and *boolean*.

The numeric primitives are: *short*, *int*, *long*, *float* and *double* and store numeric values on which mathematical operations can be performed.

The textual primitives are *byte* and *char* and store characters represented as Unicode or numbers. Comparative and arithmetic operations can be performed on variables of these types.

The boolean type can store only *true* or *false*.

### Minimum and Maximum Size

When a variable of a primitive type is declared a small space in the computer’s memory is allocated to store that variables value. The size of that space is determined by the variables type and in turn the maximum and minimum value that the variable can hold is determined by the size of the space allocated.

The smallest primitive is *byte*. It occupies 8 bits and has a value range from -128 to 127. The largest primitive is *double* and occupies 64 bits and has a range from -263 to 263-1.

The table below shows the primitive types.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Class** | **Type** | **Bits** | **Range** | | | **Default** | |
| **Integer** | byte | 8 | min | (-127) -27 | 0 | |
| max | 27-1 (127) |
| short | 16 | min | (-32768) -215 | 0 | |
| max | 215-1 (32767) |
| char | 16 | min | (\u0000) 0 | \u0000 | |
| max | (\uFFFF) (65535) 216 |
| int | 32 | min | -231 | 0 | |
| max | 231-1 |
| long | 64 | min | -263 | 0 | |
| max | 263-1 |
| **Floating point** | float | 32 | min | -231 | 0.0 | |
| max | 231-1 |
| double | 64 | min | -263 | 0.0 | |
| max | 263-1 |
|  | reference |  |  |  | Null | |
|  | boolean |  | true | false | false | |

### Integer and Floating Point Types

As you can see from the table there are two classes of numerical types: *integer* types and *floating point* types.

## Casting Primitives

Casting a primitive is about making a value of one type fit into another type and there are two types: implicit and explicit.

For example, you can cast from a *byte* to an *int* value.

byte small = 120;  
int index = small;

This type of cast is called an implicit cast because the compiler determines the type to cast to. The programmer does not need to specify the target type. This is possible because the target type’s maximum value is always larger than the maximum value of the source type. This cast is known as a widening conversion and is safe to perform as no precision can be lost in the conversion to the bigger type.

The other type of casting is called *explicit casting* and requires that you specify the target type; this is because it is a narrowing conversation from a bigger type to a smaller type and there may be a loss of precision, although not necessarily. Take a look at the following example.

A cast from a *long* to a *short*, can occur without loss of precision if the long is within the range of a short.

long result = 500;  
short total = (short) result;

The target type is specified between parentheses before the variable you want to down cast.

An implicit cast is a widening conversion from a smaller type to a larger type such as from a byte to an int and an explicit cast is a narrowing conversion from a larger type to a smaller type such as from a long to a short and requires the target type to be specified.

### Integer Overflow

You might by thinking: What happens if the source value is greater that the target type’s maximum value? Is it rounded up or down? Does it just get cut off? Or does the compiler complain about it? None of these answers are correct. What happens is something that is a little counterintuitive. The source value is wrapped around the target type’s maximum and minimum values.

Take the situation above where we cast a *long* into a *short*. If the value of the *long* is 500 and knowing that the range of a *short* ranges from -32,768 to 32,767, the value of the short will be 500.

If the value of the *long* is 32,777, just ten higher than the maximum value of a *short*, the target variable will be -32,759. As you can see the number has been wrapped around and is counting backward from the minimum value. If the *long* value was much larger the target value would continue wrapping around until it finishes.

This is one reason you need to be very careful when doing an explicit cast. The outcome might not be as expected, difficult to predict and could result in hard to find bugs. The usual solution to this is not to cast in the first place. If you have to cast you should check that the source value is within range of the target value before performing the cast.

## Different Bases

We have been using the base 10 decimal number system to set variable values. You are not restricted to this system and can use other bases to set literal values; we can use the binary, octal and hexadecimal base systems.

Binary numbers are base 2 and contain the numbers 0 and 1. They are represented in Java by a leaning 0b or 0B (zero plus letter b). For example:

byte b = 0b101;  
byte B = 0B101;

Octal numbers are base 8 and contain numbers from 0 to 8. They are represented in Java be a leading 0. For example:

int x = 0453;

Hexadecimal numbers are base 16 and contain numbers from 0 to 9 and then letters from a to f or A to F: the letters are not case sensitive. Hex numbers are indicated by a leading 0x or 0X. For example:

int x = 0x435;  
int x = 0Xa3eE;

All integer literals can be set in base 8, base 10 and base 16, but are converted to base 10 when arithmetic calculations are performed.

*Float* and *double* can be set in base 2, base 8 and base 16 but only if the number can be implicitly cast to an integer. It is not possible to mark a *float* or *double* with the letter denomination in a base 16 because it confuses the compile as *f* and *d* are letters used in the hexadecimal system. Consider the following examples:

float x = 0x1f;

In this code snippet we are trying to indicate that the hexadecimal number is the *float* value *0x1* by adding a training letter *f*, however the compiler will interpret the value as *0x1f*. The result is that you are inadvertently setting *x* to the value 31.0, rather than the intended value 1.0.

It is illegal to set a *float* or *double* variable in binary while using the trailing letter denominator. The following code does not compile:

float b = 0b101f;  
double B = 0B101d;

No such confusion exists when setting literals using the octal number system.

## Creating Objects

As you have already learnt, Java is an object orientated language. Objects are created from a blueprint which we know is represented by a class.

### Distinguish between reference and primitive variables

There are a few important distinction between primitive and object references.

Primitive references store primitive values and can only ever store primitive types. Object references store the memory address of the object and can only store memory addresses to objects

When a primitive variable is declared its initial value is 0 (or one of the default values we met earlier), an object reference’s default value is always *null*. A *null* value means that the reference variable does not point to an object in memory. When the new keyword is used to create a new object and the reference variable points to the new object the reference variable will no longer be *null*.

Primitives cannot be assigned the value *null* and object reference cannot be assigned the value 0. They are incompatible.

An important distinction is that objects have methods. These methods are called on the object instance. Primitives don’t have methods. Let’s look at an example:

1: String name = new String(“Alex Theedom”);  
2: String upperName = name.toUpperCase();

On line 1 we create a new object that contains the string “*Alex Theedom*”. It stores the address of this new *String* object in the reference variable called *name*. On line 2 we call the method *toUpperCase()* on the reference variable. This method converts all the letters to upper case and returns a new *String* object containing the text and stores the address of this new *String* object in the *upperName* variable.

## Object Constructors

When a new object is created using the *new* keyword a special type of method within the class called a constructed is executed. A constructor’s job is to set initial values and execute set up code required by the object for proper construction.

For simple objects, there is not always a business requirement to initialise the object with values or execute code, which is why the compiler will insert a default constructor that does not do anything other than to satisfy the semantic requirement for a constructor.

### Adding a Constructor

A constructer has a prescribed format and looks very much like a method, but it is not a method, and is called automatically when the object is created.

A constructor’s name must be the same as its class name. It can only be defined as having *public*, *protected*, *default* or *private* access and cannot have a return type and it can accept zero to many arguments.

Let’s have a look at an example:

public class BankAccount {}

In this code snippet we have defined a class called BankAccount but we have not defined a constructor. The compiler will insert a default constructor during compilation. A default constructor is *public*, has no arguments and does not contain any code, other than an implicit call to the method *super()*. The *super()* method calls the class’s parent constructor. We will discuss this later on in the course in **Lesson 3** **Section 2: Understand Inheritance**.

A default constructor will look like the code on lines 2 to 4.

1: public class BankAccount {  
2: public BankAccount() {  
3: super();  
4: }  
5: }

You will never see the compiler generated default constructor, but it is important to know that it is there and that it will only be inserted automatically if you do not add a constructor yourself.

### Custom Constructors

The programmer can add as many constructors as is required, however the parameter passed to the constructor must be different, so that each constructor has a different signature.

Later in the course in **Lesson 3 Section 1: Understand Methods and Encapsulation**, you will learn about more complex situations involving constructors.

## Exercises

Which of the snippets of code will not compile?

1. float x = 123
2. float x = 123.0
3. double x = 200d;
4. double x = -300;
5. float x = (float) 500;

Answers: 2

Which of the snippets of code will not compile?

1. int x = (byte) 100;
2. int x = (int) 100.0;
3. byte x = (byte) 5\_000.00;
4. int x = 100f;
5. boolean x = (boolean) false;

Answers: 4

Which of the snippets of code will not compile?

1. char x = ‘A’;
2. char x = 0Xa3eE
3. char x = ‘\u004E’;
4. char x = 340;
5. int i = 5; char c = i;

Answers: 5

Which of the following code snippets will cause a compiler error?

1. String name = new String(“Alex”); name.toUpperCase();
2. int score = 100; score.value();
3. String[] a = new String[7]; int x = a.length;
4. char c = ‘A’; c.toString();
5. new String().toUpperCase();

Answers: 2, 4

Select the code snippets that correctly define a constructor.

1. public BankAccount(){}
2. private CreditCard(String number){}
3. public void Load(int amount){}
4. public main(String… args){}
5. House(){}

Answers: 1, 2, 4, 5

# Learn to read and write to object fields (5 min)

Objects have fields that store values. We have already talked quite extensively about fields, variable scope and type. Now let’s talk about the various ways that fields can be set.

### Write (set) a value to an object field

Writing (or setting) a fields value means accessing the field and assigning it a value.

### Reading from an object field

Reading (or getting) a value from a field means accessing the field and obtaining its value.

## Instance initialiser blocks

Code written in a method is located within the code block delimiters represented by curly braces ({}). Code blocks can also stand alone and appear outside of methods.

This is what a code block might look like:

1: public class Greeting {  
2: {  
3: System.*out*.println("Hello");  
4: }  
5: }

Lines 2 to 4 define a code block that prints a *Hello* message. You might be wondering how this code gets executed. After all it has no method identifier. The answer is that it is called automatically when the class is instantiated by the creation of a *Greeting* object.

1: new Greeting();

This code block is called an *Instance initialiser block*. It is used to house code you want to execute every time the object is created. In many ways it is similar to a constructor, but is executed for every new instantiation of the class.

You can have many instance initialiser blocks and they can be located anywhere in the code, as long as they are outside of any methods.

### Static initialiser block

Just as instances can have initialiser code blocks so to can classes. These are called *static initialiser* code blocks and are declared with the *static* keyword.

static {  
 System.*out*.println("OK");  
}

These can be located anywhere within the class structure.

### Execution Order

An important characteristic of a class that has many initialiser blocks is the order in which they are executed when a new instance of that class is created.

### Order of Declaration and Usage

Initialiser blocks are often used to declare and initialise variables. Care needs to be taken with regard to the location of fields. The declaration of a field needs to appear textually before it is used. This means that a field cannot be given a value before it is declared and the order of declaration is determined by its location in the class.

In the following code snippet the code will not compile and complain about *an illegal forward reference*, on line 3.

1: {  
2: index = 3;  
3: System.*out*.println(index);  
4: }  
5: int index;

The index field used in line 3 is not declared until line 5. To resolve this issue the declaration of *index* should be done before the initialisation block.

Interestingly, the following code compiles, even though the *index* field is declared after the initialisation block.

1: {  
2: index = 3;  
3: }  
4: int index;

The key is to know when the field is used. In this example, the field is not used before it is declared and so this code will compile.

# Describe an Object’s lifecycle (10 min)

Objects have a lifecycle: they are created, they exist and then they are distroyed. At each stage of the objects lifecycle certain things can happen that you must know about.

### Object’s Lifecycle

A Java object has seven lifecycle stages that start with the creation of the object and finishes with de-allocation

## Finalise Method

We have briefly discussed the *finalize()* method but it’s worth looking at it a little more deeply. This method is called in the final stages of the object’s life and is called only once. If the object is brought back to life by allocating a strong reference it will not be called again when the object reaches this stage. If an exception is thrown while executing the *finalize()* method, the method will not be executed again either. It is possible that the method might not get called at all.

Owning to this behaviour it is highly unlikely that you will ever use this method in your projects and are generally advised against using it. The only thing you need to remember about this method for the exam is that it might never get called or get called once, but it will never get call twice.

## Garbage Collection

Now that we have covered object lifecycle and the *finalise()* method, it’s time to take a look at the garbage collector. The responsibility of the collector is to free up memory from objects that are no longer being used by the application. This is an important process that allows a program to effectively manage its memory. If objects were not garbage collected the available memory would be quickly used up and the application would run out of memory and stop executing.

Unlike other programming languages, memory management in Java is largely automatic. The programmer does not need to do anything to provoke it. The collector has algorithms for selecting objects that are no longer required by the program and releasing their memory allocation.

However, you can suggest to the garbage collector that now is a good time to consider starting collection by calling the *gc()* method on the *System* class.

System.*gc*();

A call to the *gc()* method does not force the collection of garbage, it’s just a suggest that it should expend effort toward recycling unused objects.

### Object and Object references

To fully understand how the garbage collector determines which objects are collectable you must understand the difference between the object reference and the object itself. We have touched on this topic lightly already and you should know that an object reference (aka instance reference/instance variable) contains the memory address of the actual object, not the object itself. It just refers to the object’s location on the heap. This is an important distinction because when an object no longer has a reference to it, the object becomes eligible for garbage collection.

Let’s look at an example.

String name = new String(“Alex Theedom”);

**The Heap**

[object reference]  
name

In this diagram the *String* object is referred to by the object reference *name*.

Both can exist independently of each other. An object can be created without a reference: *new String(“Alex Theedom”);* and a reference can be created without referring to an object: *String name;.* This is not useful code to write because you want to create an object and use it, and an object is only usable if you have a reference to it. Nevertheless, this does demonstrate that they are two different but connected concepts.

The garbage collector will collect objects that are no longer referred to by an object reference. Objects can be dereferenced in a variety of ways. Let’s examine two of them.

1. The instance reference is nulled
2. The instance reference is reassigned to another object

# Use primitive wrapper classes (10 min)

Java SE 5 saw the introduction of primitive wrapper object classes. These are classes that are the object equivalent of a primitive type but with the added functionality of methods that perform conversations and comparisons.

Each primitive has its equivalent object wrapper class. For example the code snippet below shows two ways to set an integer value.

1: Integer score = new Integer(100);  
2: int result = 100;

Line 1 creates an integer object with an internal value of 100, while line 2 set a primitive with the same value.

Below is a table of the wrapper class mapped to their equivalents primitive types.

|  |  |
| --- | --- |
| **Wrapper Class** | **Primitive Type** |
| Byte | byte |
| Short | short |
| Character | char |
| Integer | int |
| Float | float |
| Long | long |
| Double | double |

Note the difference in the names of the *int* and *char* wrapper classes.

### Using wrappers

Wrapper object have a few very useful methods. Let’s look at a few of them.

### Conversion to the same or difference primitive type

All wrapper objects have methods that convert the object type to one of the primitive types. If the target type is smaller than the source type a narrowing conversion is applied to the value, in the same way that an explicit cast narrows to a smaller number.

1: Integer score = new Integer(1\_000\_000);

2: int total = score.intValue();  
3: byte result = score.byteValue();

### String parsing to a primitive/wrapper type

Each wrapper class (except *Character*) has a *parseXXX(String)* method that parses a *String* to the wrapper type’s primitive equivalent.

1: short lucky = Short.*parseShort*("777");  
2: long lotto = Long.*parseLong*("2311234534");

There is an equivalent method that returns an instance of the wrapper type instead of the primitive type.

1: Short lucky = Short.*valueOf*("777");  
2: Long lotto = Long.*valueOf*("2311234534");

### Conversion of a wrapper type to its String equivalent

A wrapper type can be converted to a *String* with the *toString()* method which returns a *String* object containing the primitive number as a *String*.

1: Double price = new Double("5\_300.45");  
2: String cost = price.toString();

These are just a few of the methods available for conversions. Take time to look at the wrapper classes and discover what other methods these classes have to offer.

It is important to note that if you attempt to pass an argument to one of the wrapper conversion methods that expects a number and it cannot be successfully converted to the required number, the methods will throw an exception called *NumberFormatException*. We will look at exception and exception handling later on in the course in the **Lesson 2 Section 3: Understand Exception Handling**.

## Autoboxing

Primitives are automatically converted to their respective Wrapper type and vice versa in certain situations. Take a look at the following code snippet:

1: Integer score = 5;  
2: int result = new Integer(5);

In line 1 the primitive number 5 is autoboxed (automatically wrapped) to an Integer object so that is can be assigned to the *score* Integer variable and on line 2 the Integer wrapper object whose value is 5 is unboxed (automatically converted) to a primitive *int*.

The boxing and unboxing is done at compile time by the compiler. Later on in this course we will see examples of using autoboxing in **Lesson 3 Section 1: Understand Methods and Encapsulation**.