Féidearthachtaí as Cuimse
Infinite Possibilities

Bits and pieces: Data types, wrappers, operators, constants

Object Oriented programming

OLLSCOIL TEICNEOLAÍOCHTA BHAILE ÁTHA CLIATH
TU DUBLIN
TECHNOLOGICAL UNIVERSITY DUBLIN

Data types

- Using a variable in java.. You must declare what type of data can contain..
  - int, char etc..
  - e,.g
    - `String studentNumber = "C488573"`

- A primitive type is a data type predefined by the language and is named by a reserved keyword....
- 8 of them in java

primitive data types (java)

- char (16 bits) a Unicode character
- byte (8 bits)
- int (32 bits) a signed integer
- short (16 bits) a short integer
- long (64 bits) a long integer
- float (32 bits) a real number
- double (64 bits) a large real number
- boolean - true or false

A note on boolean

- boolean
  - values are true or false (keywords)
  - Used with control statements e.g. while, if and conditional expressions
  - e.g. `while (fileEmpty)`

Primitive ↔ Wrapper Classes

- Each primitive data type also has a **class** equivalent.. called a "wrapper" class: e.g.
  - e.g. Integer class (equivalent to "int") etc.

```
Integer studentNumber = 88833385;          // autoboxing
// or:
Integer studentNumber = Integer.valueOf(88833385);
```

Primitive Data Types	Wrapper Classes
int	Integer
short	Short
long	Long
byte	Byte
float	Float
double	Double
char	Character
boolean	Boolean

Which to use?

Convention is to use **primitives:**

- are faster on performance; less error prone

- However: Parts of the JAVA API only allow objects as parameters
  - e.g. ArrayList . if needed to store/ sort a list of numbers)
  - Can use **wrapper** classes classes to convert primitive data type to an object

Primitive vs Reference types – difference?

e.g.

```
int accountNum = 9274;     // primitive
```

Versus

```
Integer accountNum = new Integer(9274)
                                  // reference (deprecated now)
```

Versus

```
Integer accountNum = 9274;    //autoboxing
```

Versus

```
Integer accountNum = Integer.valueOf(9274);
```

Is `accountNum` the same thing in both cases?

Similarities? Differences?

Primitive vs Reference types

Primitive variables store the actual values.

Reference variables store the addresses of the objects they refer to. Objects have behaviour..

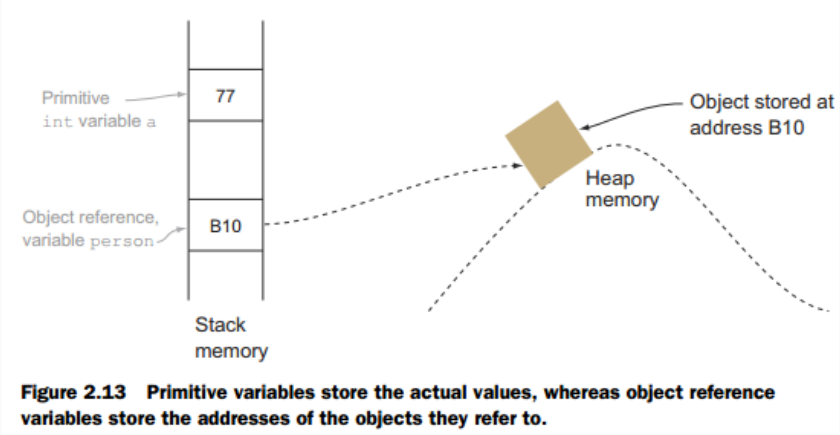


Figure 2.13 Primitive variables store the actual values, whereas object reference variables store the addresses of the objects they refer to.

When you copy a primitive → you copy the value.
When you copy a reference → you copy the address, so both variables refer to the same object in memory.

Primitive vs Reference types

Attributes of a class can primitive and/or references types e.g.

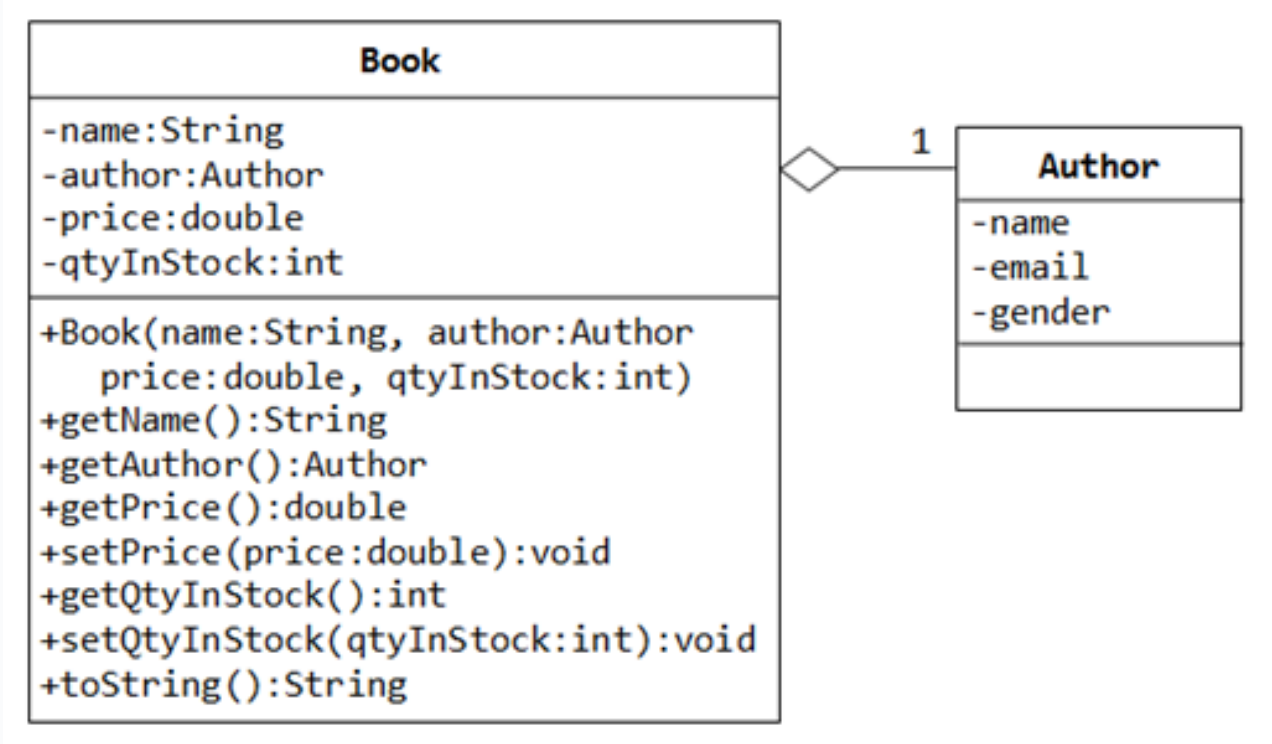
```
public class Book
  //
  double price;
  String name;
  Author author;
  int qtyInStock;
```

Primitive vs Reference types

Class Book **is "composed" of** an Author object (and also has a price, name, qtyInStock)

```
public class Book
  //
  double price;
  String name;
  Author author;
  int qtyInStock;
```

UML notation for composition



Operators

- Additive
  + -
- Multiplicative
  * / %
- Equality (tests)
  == !=
- Assignment operators
  = += -= *= /= %=

Operators

- Relational operators
  `<  <=  >  >=`
- Increment operators (postfix and prefix)
  `++  --`
- Conditional operator (short cut for if/else
  `?:` e.g. max = (a > b) ? a : b;
- String concatenation
  `+`

Usage

- int a = 5, b = 3;
- int sum = a + b; // arithmetic
- boolean test = a > b; // relational
- a++; // increment
- a += 2; // assignment
- boolean flag = (a > b) && (b > 0); // logical
- int max = (a > b) ? a : b; // conditional
- String msg = "Result: " + sum; // String concatenation

- These operators mainly work on **primitive data types**,
  except + for **Strings** and == for **object references** (which checks if they are the same object in memory).

Overview

primitive type stores a simple value directly, like a number or a boolean.

reference type stores a reference (address) to an object in memory, not the actual value. Has behaviour and methods.

Final

- final for Constants

  final on a **variable** means the value **cannot change** after it is assigned.

  final int MAX_MARK = 100;

  //Treated as a constant

  //Trying to change it causes an error:

Final on methods

```
class Ball {
    public final void toss() {
        System.out.println("Ball
toss");
    }
}
```

Subclasses **cannot** override:

Why Use It?

To lock behaviour

To protect logic in inheritance

To prevent accidental overrides