Types I – getting started

CONTENTS

- **Objectives**
  - To understand how to define types and create objects using Java
  - To grasp the concept of reference type behaviour
- **Contents**
  - OO Fundamentals – abstraction and encapsulation
  - Defining reference types – keyword class
  - Creating objects (instances) & seeing reference type behaviour
- **Hands on Labs**

OO Fundamental – Abstraction

- **Ability to represent a complex problem in simple terms**
    - In OO, creation of a high level definition with no detail yet
        - Add detail later in the process
    - Factoring out common features of a category of data objects
- **Stresses ideas, qualities & properties not particulars**
    - Emphasises what an object is or does, rather than how it works
        - Primary means of managing complexity in large programs
- **Students (instances of type Student) attend a Course**
    - Have ‘attributes’- name, experience, attendance record
    - Have ‘behaviour’ - listen(), speak(), takeBreak() doPractical()
    - Are part of ‘relationships’
        - A student ‘sits on a’ course, a course ‘has’ students

OO Fundamental – Encapsulation

- **Hiding object’s implementation, making it self-sufficient**
- **Process of enclosing code needed to do one thing well**
  - Plus all the data that code needs in a single object
  - Allows complexity to be built from (apparently) simple objects
    - Internal representation & complexities are hidden in the objects
    - Users of an object know its required inputs and expected outputs
    - Substantial benefits in reliability, maintainability and re-use
- **Objects communicate via messaging (method calls)**
  - Messages allow (receiving) object to determine implementation
  - Sender does not determine implementation for each instance

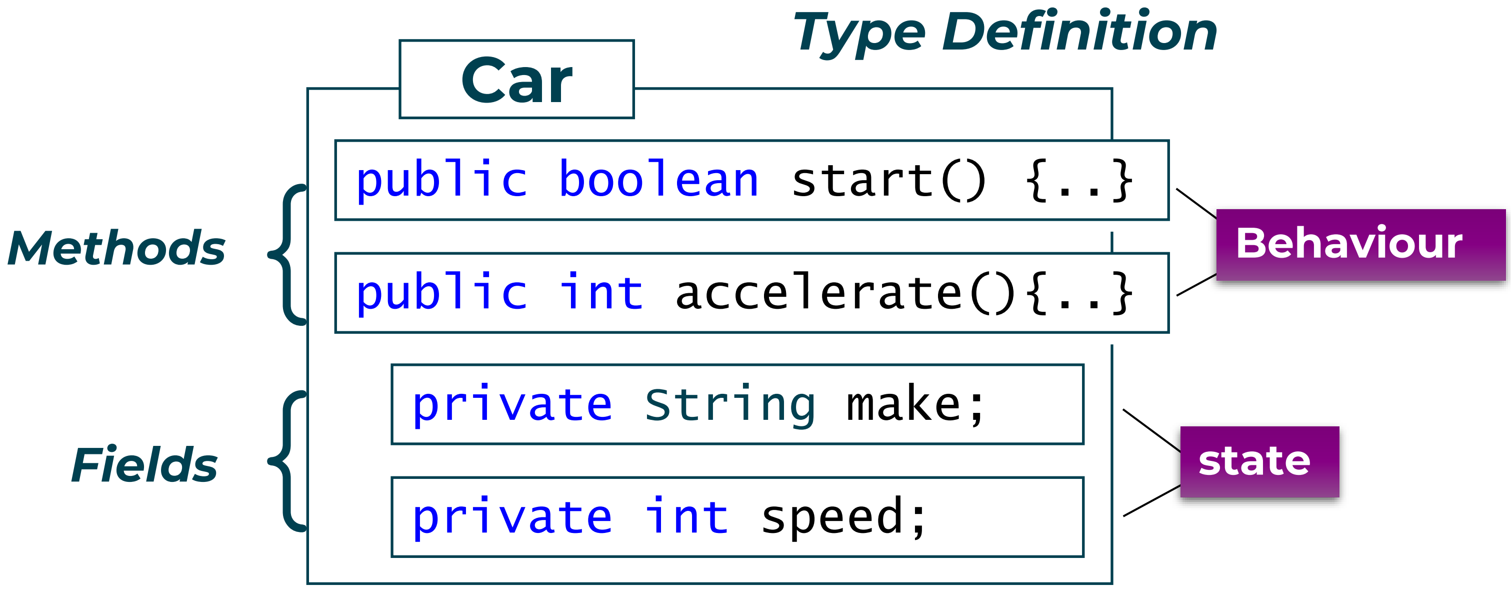
```
for(Student s : myStudents){ s.doNextLab(30);}
```

“I tell you how long you have, you sneak in the ‘comfort’ breaks”

What is an OO data type?

class definition is a blueprint, a ‘plan’ for making *objects*

- Fields - Constituent data parts. Hold state
- Methods - Functions that define behaviour



```
public boolean start() {..}
public int accelerate(){..}
private String make;
private int speed;
```

Public members form the object interface

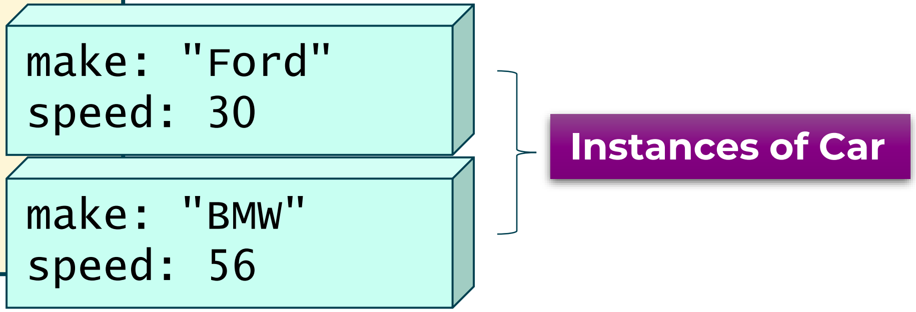
Classes and Objects

- **Objects are unique instances of a class with own state.**

Blueprint

```
public class Car {
   public String make;
   public int speed;

   public void start() {
        print("Car starting");
   }

   public void stop() {
      speed = 0;
   }

   public void accelerate() {
      speed += 2;
   }
}
```

```
public static void main(…) {

   Car car1 = new Car();
   Car car2 = new Car();

   car1.make = "Ford";
   car2.make = "BMW";
   car1.speed = 30;
   car2.speed = 56;
}
```



```
make: "Ford"
speed: 30
```

```
make: "BMW"
speed: 56
```

Instances of Car

Getters and setters

- **Do not expose state**

```java
public class Student {
   private String name;
   private int age;
}
```

How to set the name and age?

```java
public static void main(…) {

   Student stu = new Student();

   stu.name = "Bob"; ×
   stu.age = 25; ×

}
```

Getters and setters

- **Do not expose state**

```java
public class Student {
   private String name;
   private int age;

   public String getName() {
        return name;
   }
   public void setName(String name) {
        if(name.length() > 1)
               this.name = name;
   }

   public int getAge() {
        return age;
   }
   public void setAge(int age) {
        this.age = age;
   }
}
```

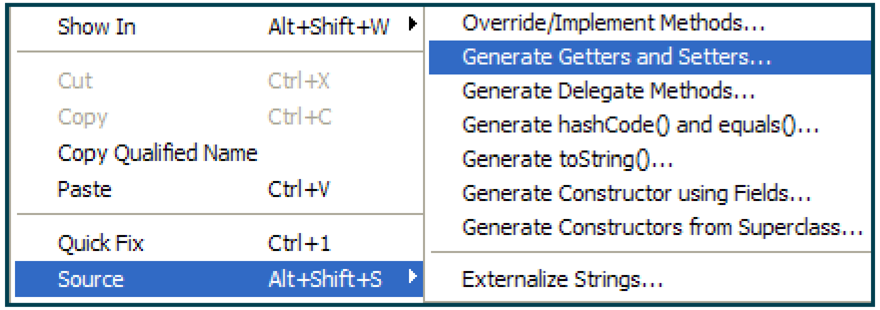
```java
public static void main(…) {

   Student stu = new Student();

   stu.name = "Bob"; ✗
   stu.age = 25; ✗

   stu.setName("Bob"); ✓

   stu.setName("B"); ✓
}
```

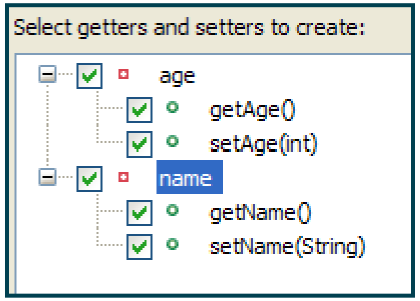
But name is not changed

Encapsulating via the IDE

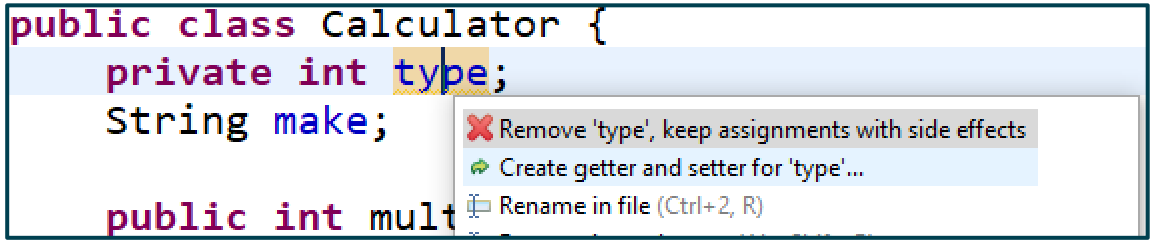
IDE will write getters and setters for you based on fields defined

- Can write a ‘classful’ of methods in seconds
- Right click anywhere in editor pane for..


Show In Alt+Shift+W
Cut Ctrl+X
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Quick Fix Ctrl+1
Source Alt+Shift+S
Override/Implement Methods...
Generate Getters and Setters...
Generate Delegate Methods...
Generate hashCode() and equals()...
Generate toString()...
Generate Constructor using Fields...
Generate Constructors from Superclass...
Externalize Strings...



- If you have focus on an individual field then press **Ctrl-1**



```
public class Calculator {
    private int type;
    String make;

    public int mult
```

Remove 'type', keep assignments with side effects
Create getter and setter for 'type'...
Rename in file (Ctrl+2, R)

Object Construction

- **Let's consider two classes and the two instances created**

```
public class Car {
  private int speed;
  private String make;
}
```

```
Car myCar = new Car();
```

What make is this?
What is its speed?

```
public class Account {
  private int id;
  private String owner;
}
```

```
Account myAccount = new Account();
```

What is the id of this account?
Who owns it?

We need a constructor

Constructor

```
public class Account {
  private int id;
  private String owner;

  public Account (int id, String owner) {
      this.id = id;
      this.owner = owner;
  }
}
```

The same name as the class. No return value. Not even void

```
Account myAccount = new Account(123, "Bob");
```

✓

```
Account myAccount = new Account();
```

✗

The default (parameter-less) constructor does not exist. To create an Account you must provide the ID and the owner's name

Object Construction - Overloading

- **Overloading provides alternative ways for creating an instance**

```
public class Account {
  private int id;
  String owner;

  public Account (int id, String owner) {
      this.id = id;
     this.owner = owner;
  }

 public Account (int id) {
      this.id = id;
     this.owner = "June”;
  }
}
```

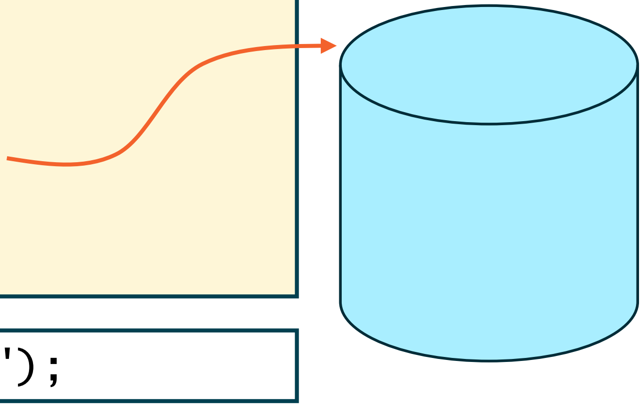
```
Account myAccount = new Account(123, "Bob");
```

```
Account myAccount = new Account(123);
```

Object Construction - Overloading

- **Overloading provides alternative ways for creating an instance**

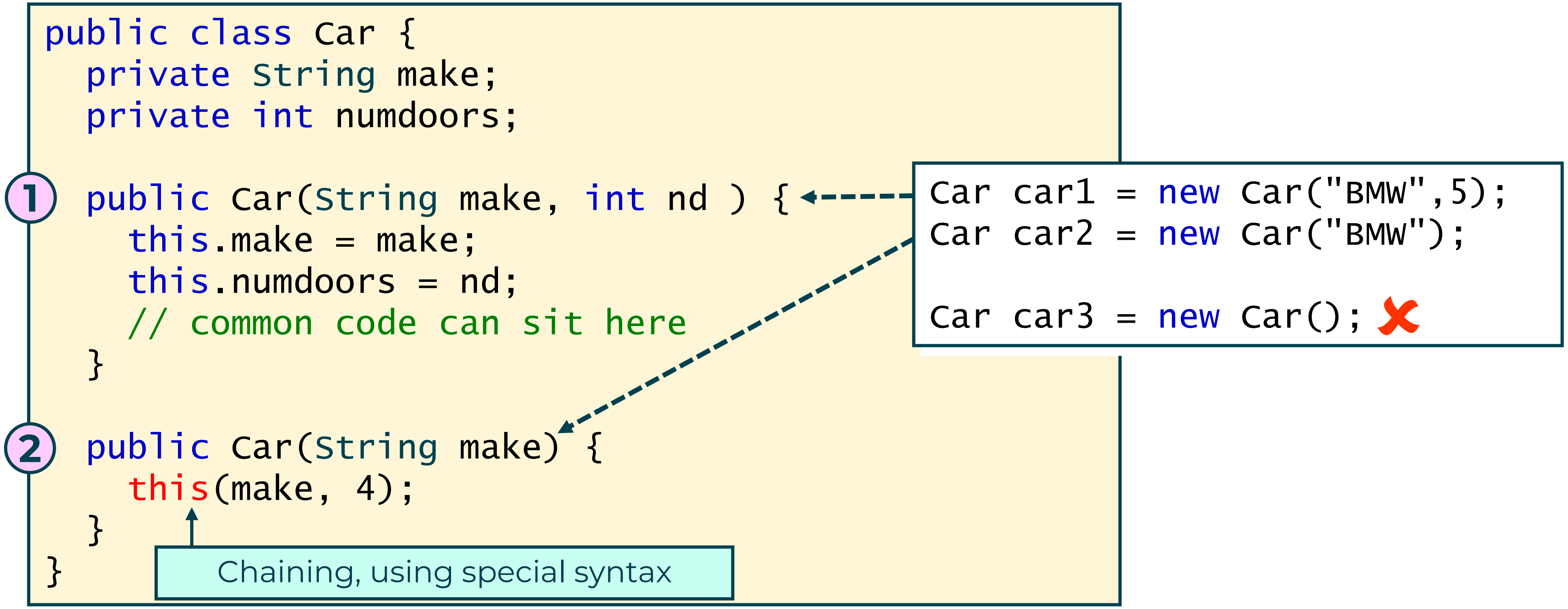
```
public class Account {
  private int id;
  String owner;

  public Account (int id, String owner) {
      this.id = id;
     this.owner = owner;
  }

 public Account (int id) {
      this.id = id;
     this.owner = getOwnerById(id);
  }
}
```



```
Account myAccount = new Account(123, "Bob");
```

```
Account myAccount = new Account(123);
```

Constructor Chaining Example



```java
public class Car {
  private String make;
  private int numdoors;

  public Car(String make, int nd ) {
    this.make = make;
    this.numdoors = nd;
    // common code can sit here
  }

  public Car(String make) {
    this(make, 4);
  }
}
```

```java
Car car1 = new Car("BMW",5);
Car car2 = new Car("BMW");

Car car3 = new Car(); ✗
```

Chaining, using special syntax

2 **Overloaded .ctor chains to the other ctor, must be 1st statement**

The null Reference – Setting and comparing

```
public Car getPoolCar() {
        Car aCar = null;
        // attempt to get a Car from a pool of cars
        return aCar;
}
```

aCar does not reference an object

```
public void hireCar() {
        Car car1 = getPoolCar();

        if (car1 != null) {
                // Drive the car away
        } else {
                print("No car available");
        }
}
```

Can compare an object reference with `null`

Review

- OO concept of defining a type
- Defining ref types – keyword class
- Understanding the concept of an ‘object’ reference

Hands On Lab

- Creating and using reference types
- Passing reference types to a method

Arrays – revisited

All array variables are reference variables

- pass a ref to any array – by value

```
public class Car {...}
```

Assuming this class defined

```
int num = 0;

int[] nums1 = new int[3];

int[] nums2 = { 3, 5, 7, 9};

Car[] cars1;

Car[] cars2 = new Car[3];

Car[] cars3 = {new Car(),
               new Car(),
               new Car()};
```

‘num’ is a value type = 0

‘nums1’ is a ref type, 3 zeros in

‘nums2’ is a ref type, .length = 4

‘cars1’ is an un-initialised reference variable

‘cars2’ is a reference variable, .length = 3
but contains 3 nulls & no cars!!

‘cars3’ - a reference to an array of car references

```
processIntArray(nums2);
processCarArray(cars3);
```

Types in the Java runtime

- **Java runtime supports 2 sorts of ‘type’ - value & reference**
  - Here we focus on reference types
    - Exhibit ‘reference type’ behaviour, objects meant for ‘sharing’
  - Main way to define a reference type – use keyword `class`

- **Behaviour of classes**
  - Support inheritance (by default)
  - Objects only created via keyword `new`
  - Reference (like myCar) can be passed to methods
    - If ‘local’ to a method, on stack, deleted at end of method

```
public class Car { .. }
```

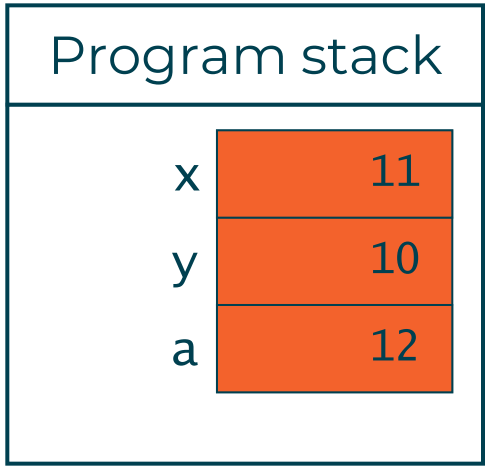
```
Car mycar = new Car();
```

  - Object lives on managed heap – get garbage collected
  - Examples `Car, Button, String`

Classic Value Type Behaviour – reminder!

```java
public class Program {
   public static void main(…) {
      int x = 10;
      int y = x;
      x++;
      System.out.println( x );
      System.out.println( y );
      foo( x );
      System.out.println( x );
   }

   public static void foo( int a )
   {
      a = a + 1;
   }
}
```

Step



Program stack

x	11
y	10
a	12

Console

```
C:\> Program
11
10
11
```

Reference Type Behaviour – different!

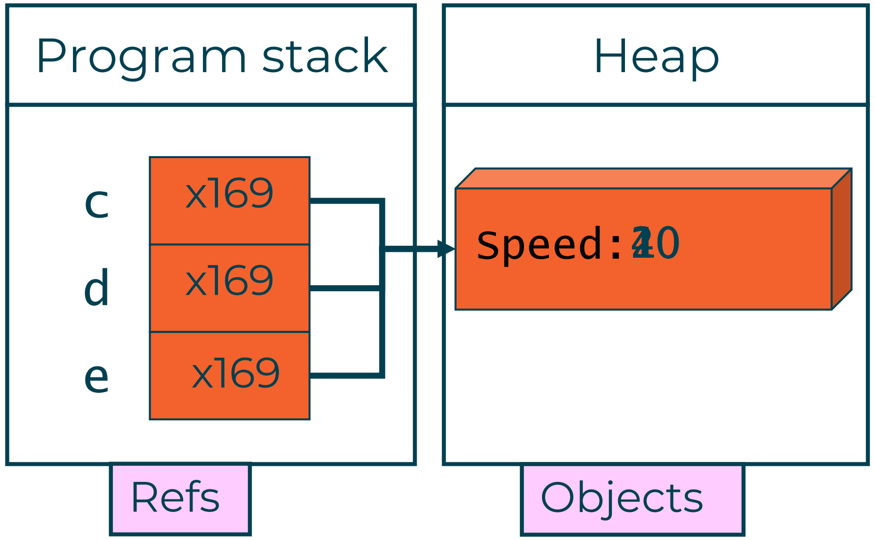
```
public class Program {
 public static void main(…) {
  Car c = new Car();
  c.accelerate(10);
  Car d = c;
  d.accelerate(10);
  System.out.println(c.getSpeed());
  System.out.println(d.getSpeed());
  foo( c );
  System.out.println(c.getSpeed());
 }
 public static void foo(Car e)
 {
  e.accelerate(20);
 }
}
```

Copy of reference passed!

```
public class Car {
  . . .
}
```

Functionality of accelerate() & getSpeed()

Step



Console

```
C:\> Program
20
20
40
```