Introduction to Programming

9. Types, Classes and Objects
Part 1

Types revisited

- A *type* is a set of values, and a set of operations that can be applied to those values
- For the primitive types, and for the type, String, the values can be written directly using literals
  - `"This is a String literal"`
  - `7.01` // a **double** literal
  - **`false`** // a **boolean** literal

The primitive numeric types

- Integer types
  - **byte:** 1 byte values from -128 to 127
  - **short:** 2 byte values from -32768 to 32767
  - **int:** 4 byte values from -2147483648 to 2147483647
  - **long:** 8 byte values from -9223372036854775808 to 9223372036854775807
- **char**
  - character is represented by its position number in the 2 byte Unicode character set – **char** is an integer type under the hood
    - **char** is only numeric type with no negative values, so can hold values from 0 to 65535
- Floating point types
  - **float:** 4 byte values, approx 7 digits precision, $\pm 10^{38}$
  - **double:** 8 byte values, approx 15 digits precision, $\pm 10^{308}$

Type conversions revisited

- Automatic "widening" conversions
  - Can assign a value of one primitive numeric type to a variable of another primitive numeric type
  - If the range of values of the type being assigned to is wider than range of values of the type being assigned, the value is converted (promoted) automatically

    double velocity = 200; // **int** value, 200, converted to **double**
  - Figure below lists the types in order of how wide their range of values is and says you can assign a value of any type shown to the left of a type to a variable of that type (e.g. can assign a **short** value to a **float** variable as **short** is to the left of **float**)

$$\begin{tikzcd}
\textbf{byte} \arrow[r] & \textbf{short} \arrow[dr] & & & & \\
 & & \textbf{int} \arrow[r] & \textbf{long} \arrow[r] & \textbf{float} \arrow[r] & \textbf{double} \\
 & \textbf{char} \arrow[ur] & & & &
\end{tikzcd}$$

Type conversion continued…

- Possible loss of precision with widening conversions
  - **float** has about 7 significant figures, **double** has about 15; **int** and **long** values may have more significant figures than this, so automatic conversion can lose precision

```
float real = 1234567890123456L;
TextIO.putln(real); // prints 1.23456795E15
```

- Do not confuse precision with magnitude

Casting Primitive Values

- To convert a numeric value from a type with a wider range of values to one with a narrower range of values ("narrowing" conversion) you usually need to use a *cast* so the following is NOT legal:

```
int metre = 100.0; // won't compile: assigning double to int
```

- As the range of **double** is wider than that of **int**, need to cast the value being assigned

```
int metre = (int) 100.0; // fine now, (int) casts 100.0 to 100
int kilometre = (int) (1000.0*metre);  // cast for double*int
kilometre = 1000*metre;  // no cast needed as int*int
```

Loss of information when casting

- When casting a real value to an integer value the fractional part is discarded

```
int pi = (int)Math.PI; // pi equals 3
```

- Beware of casting a value out of the range of the target type!

```
byte b = (byte)1000; // max byte is 127
TextIO.putln(b);          // prints -24
```

Restrictions with simple types

- A number of requirements cannot be met with just the primitive types
  - Collections: the need to define multiple items of the same type
    - e.g. Marks for every student in a class of 40
  - Need to restrict the range of values in a simple variable
    - Types with relatively few valid values (e.g. months, days, planets in the solar system, chemical elements)
  - Composite things: need to combine several values to create a representation of a real-world thing
    - e.g. NHS Patient record, Train route details (stations, times, fares, operator etc.)

Data in Java Applications

- Java provides
  - **Classes** to combine data items and encapsulate them with the operations that act on that data
    - Class contains a list of member variables and methods that access them
    - Member variables may belong to primitive types or may refer to instances of other classes (including arrays or enums) or the same class
  - **Arrays** to provide multiple storage slots under one name
    - Use [ ] brackets in declaration to specify that the object is an array
    - Use [ ] brackets in statements to access elements – e.g. data[6]; (7th)
    - Note that an array is a kind of class
  - **Enumerations** (**enum**) to allow definition of restricted ranges
    - Allocates a name to each of a list of values (planets, months etc.)
    - Note that an **enum** is a kind of class

Java classes

- A class declaration defines a new *reference type*
- The class name can be used to declare variables of that reference type (just as the primitive type names can be used to declare variables of primitive types)
- The values stored in reference type variables are references that allow access to instances of the class

Example

- Simplified Student record (see textbook page 171) – holds data about students on a course with three tests

```
public class Student {
    public String name; // Student's name.
    public double test1, test2, test3; /* Grades on
                                          three tests. */
   // Calculate the average test grade
    public double getAverage() {
        return (test1 + test2 + test3) / 3;
    }
} // end of class Student
```

Notes on Student Example

- No variables or methods in the Student class are declared as **static**
  - So only purpose of the class is to create Student objects
- Each Student object (or instance) has its own name and three test grades
  - The variables that hold this information are called ***instance variables***
- Each Student has its own method to calculate its average grade
  - Note the method has no parameters but directly accesses the three instance test grade variables
  - getAverage() is an example of an ***instance method***

Students as Objects

- In object-oriented programming, an *object* has
  - State – this is described by the values of its instance variables
  - Behaviour – this is described by the methods that it defines
    - Calling the methods may or may not change the state of the object
- One can think of each instance of the class Student as an object
  - currently a poorly designed one (other classes can change a Student’s state without calling its methods, so the Student class has no control over the state of its instances)

Constructors

- Are used to create instances of classes, and to initialise the state of the instance
- Like a method, but has no return type and must have the same name as the class
- A class can have several constructors as long as the argument list is different for each (overloading)
- The compiler will create a constructor with no arguments for any class that does not declare any constructors
  - Constructor with no arguments is called a “default constructor”
  - The compiler provides the Student class with a default constructor as the class does not declare one

Example continued…

- A Java application that uses the Student class (see textbook page 172)

```
public class Application {
  public static void main(String[] args) {
    Student std, std1, std2, std3;// declare Student variables
    std = new Student(); /* create a Student object and store
                            the reference to it in std */
    std1 = new Student(); // create a second Student object
    std2 = std1; // std2 and std1 both refer to second Student
    std3 = null; // store a null reference in std3
    std.name = "John Smith";
    std1.name = "Mary Jones";
  }
}
```

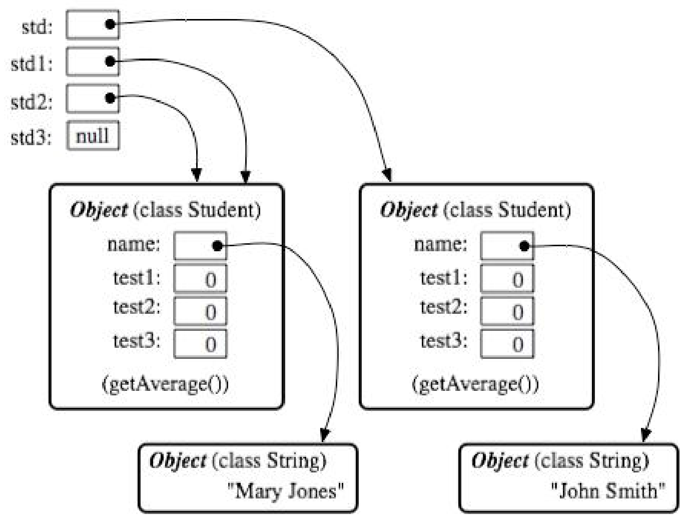
Notes on Example

- Application calls Student constructor to create two Student objects
- Value of std3 is **null**
  - Can check if a variable equals **null**

    `if (std3 == null) { ... }`
  - Runtime error if attempt made to access an object using a null reference

```
std3.name = "John Jones";
        // NullPointerException!
```

Effect of running Application
(from page 173 of textbook)



Notes on reference types...

- For reference types (objects) in general
  - The value of any variable that is not of a primitive type is either **null** or is a reference to an object
    - Note: the value of a variable of a primitive type is a value of that type (**false**, 42, 2.718, 'a', *etc*)
  - An object has to be created explicitly, usually by calling a constructor
    - From above 2 points: declaring a variable of the object's class does NOT create an object
  - More than one variable can refer to the same object
    - assignment copies the reference and not the object

Also note from example...

- Instance variables are automatically initialised when the instance is created
  - test1, test2, test3 are all initialised to 0.0
    - Any other primitive numeric fields would also be set to zero
    - Any **boolean** fields would be set to **false**
  - name, a variable belonging to a reference type, is initialised to **null**
    - main() updates the value of name, but does not update any of the test scores

Some consequences…

- The meaning of assignment
  - "="
    - Recall that the assignment operator copies a value into a variable

      ```
      std2 = std1;
      ```
    - If the right-hand-side of the assignment statement (e.g. `std1` in the above assignment statement) is the name of a variable then the value that is copied is the value that is stored in the variable
      - for a reference type, this is a *reference to an object* and NOT the object itself

Some more consequences…

- The meaning of equality
  - "==" and "!="
    - These operators compare whether two values are equal or not equal. If the values are references, the comparison returns whether the references are to the same object, NOT whether the two objects have the same state
    - To compare whether two objects have the same state, use the `equals()` method

```
if (std1.name.equals("Mary Jones"))…
```

For primitive types

- Variables of primitive types contain values of the primitive type (such as 0, 1.4, -6, false, 'r')
  - Any changes made by a method to a formal parameter of a primitive type have no effect on the actual parameter (as the formal parameter just contains a copy of the value of the actual parameter)
  - Using "==" and "!=" to compare primitive values is fine, it is the primitive values which are being compared
  - "a=b;" copies value of *b* into *a*, for primitive types any subsequent changes to *b* do not affect *a* or *vice versa*
  - Any variable of a primitive type that is declared as **final** is a constant and cannot be changed in Java

For reference types

- Variables of reference types contain references to objects of the type (or are **null**)
  - Methods cannot change *which object* the actual parameter of a reference type refers to but *may* use the reference to change the object's state (that is, to change the values of the instance variables in the object)
  - Using "==" and "!=" to compare reference values is good for checking if a value is **null** but you should use the `equals()` method to compare the state of two objects
  - "a=b;" only copies the reference in *b*, both *a* and *b* refer to the same object and changes to the object's state affect both
  - A variable of a reference type declared as **final** always refers to the same object, but **final** does not prevent the state of the object referred to from being changed

Classes

- Next week we will return to the subject of classes and how to use them
- For the rest of this lecture, will have a first brief look at array types
  - In Java, an array type is a class

Arrays in Java

- Most programming languages provide array types – Java is no exception
- In Java, arrays are classes (reference types)
- An **array object** is an *indexed list* of elements of the same type (a primitive type or a reference type)
- The **array type** name is written as

  elementType[]
- The type name can be used to declare variables (including parameters to methods and constructors)

Declaring an array variable

- To declare an array variable you use the array type name

```
int[] list;  /* can refer to arrays of
                int values /*

String[] names;  /* can refer to arrays
                    of String values */
```

Array objects

- The element type of an array can be
  - a primitive type *or*
  - a reference type (*eg* the parameter to main() is an array of String, which is a reference type)
- When an array object is created, every element of the array is given a default value
  - 0 for numeric types
  - **false** for **boolean**
  - **null** for reference types
- The syntax to call an array constructor is:

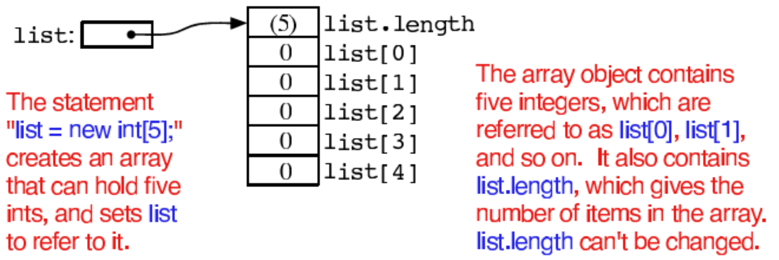
  new *ElementType* [*arrayLength*];

Creating an array object

- To create an array object, call the constructor for the array type and specify how many elements the array should contain

```
int[] list;
list = new int[5]; /* create an array
       object with five int elements */
```

Array of primitive types (Eck, page 319)



list:
(5)
0
0
0
0
0
list.length
list[0]
list[1]
list[2]
list[3]
list[4]
The statement "list = new int[5];" creates an array that can hold five ints, and sets list to refer to it.
The array object contains five integers, which are referred to as list[0], list[1], and so on. It also contains list.length, which gives the number of items in the array. list.length can't be changed.

Arrays in Java continued

- The first array element has index 0
  - An array of 5 elements has elements [0] to [4]
- Each instance of an array type has a field called *length* = the number of array elements
- Array instances (objects) are created using an array constructor, once created an array object cannot change in length
- An array reference variable refers to an array object or has value **null**

Accessing an array element

- Can think of each array element as a variable of the element type
- Use its index number in square brackets to access the element

```
int[] list = new int[5];
list[0] = 1;  // regard list[0] as an int variable
for (int i = 1; i < list.length; i++) {
    list[i] = list[i-1]*2;  // list[i] as a variable
}
```

Notes on accessing elements

- So you can think of an array as a collection of variables whose type is whatever the element type of the array is
- You access the element using its index
  - To get its value
  - To update its value
- The loop on the previous slide set the value of each array element to be twice that of the element before it

Another example

- Here is some code that reads the length of an array and the values of its elements from the keyboard

```
TextIO.put("Enter the length of the array: ");
int length = TextIO.getlnInt();
int[] list = new int[length];
for (int i=0; i<list.length; i++) {
   TextIO.put("Enter value for element " + i + ": ");
   list[i] = TextIO.getlnInt();
}
```

Final comments on arrays

- We will come back to arrays in a later lecture and look at some common algorithms involving arrays
- Arrays are a very useful way of handling a collection of items of the same type
- Arrays are the subject of chapter 7 of the book

For next week...

- Reading
  - Read **sections 5.1 and 5.2** of the textbook (this covers a little bit more that what we have covered in the first part of this lecture and gives some more examples and explanation)