

STRUCTURE OF AN OBJECT

Table of Contents

STRUCTURE OF AN OBJECT	2
PROCEDURAL VERSUS OBJECT-ORIENTED PROGRAMMING	2
CLASSES VERSUS OBJECTS.....	3
THE STRUCTURE OF A CLASS	5
THE STRUCTURE OF AN OBJECT	5
<i>The Id</i>	6
<i>The Type</i>	6
<i>Fields</i>	6
<i>Methods</i>	7
THE NEW OPERATOR.....	8
THE INSTANCEOF KEYWORD.....	9
POINTERS.....	9
GLOBAL VARIABLES.....	10

Structure of an Object

Keywords: `class`, `new`, `instanceof`

The class forms the foundation of object-oriented programming. A class can be viewed as an abstract data type that has been enhanced by the addition of object-oriented functionality.

Procedural versus Object-Oriented Programming

Generally, programming involves the manipulation of some form of data. The data is represented by a construct that captures the attributes or features of the data in question. For example, employee ID or the type of car. The manipulation of the data is provided via a set of operations. These operations generally consist of primitive operations (low level and built into the programming language) and functions (custom operations built by the programmer). For example, in C++ and Java, the `+` represents the primitive addition function and `copy()` represents a function created by the programmer. In other languages there may be no separation between primitive and customized operations.

The important point to note is that a programming language provides mechanisms for the manipulation of data. Therefore, the way the data is represented and manipulated by a programming language defines its methodology.

Procedural programming consists of data and a set of functions that operate on that data. The functions and the data are kept separate where the data is either global or passed to the functions (as actual parameters). Functions can call other functions, which results in a hierarchical type of structure, as shown in figure 1.

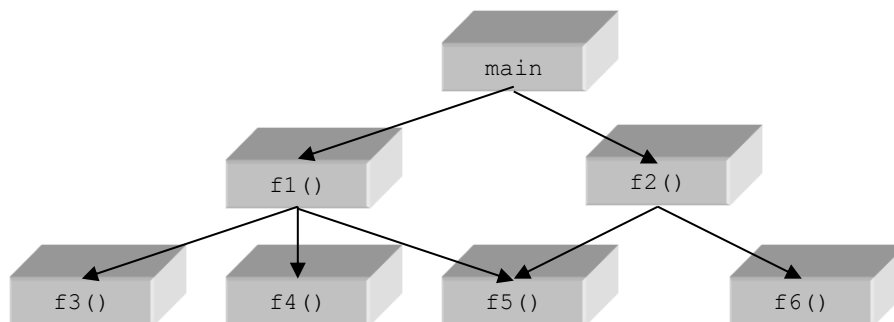


Figure 1

Object-oriented programming also consists of data and relevant functions, however, the data and the functions that act on it are encapsulated in a package called an object. As a result, an object carries both its data and its functions, making it an independent entity with loose coupling between it and other objects. For an object to utilize the functionality of other objects, messages are used. In other words, objects communicate with each other via messages. This results in a program that resembles a connected network as shown in figure 2.

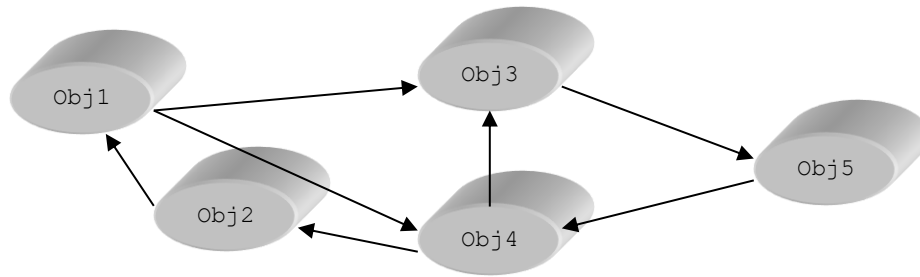


Figure 2

The major benefit of object-oriented programming is that object modeling matches more closely to that of the real world. Also, it encourages and can simplify software reuse. Error tracking, software modification and upgrades are also somewhat simplified when these concepts are used correctly.

Finally, it should be remembered that procedural programming and object-oriented programming are very different in terms of design style and require entirely different ways of thinking. Consequently, you should not apply procedural design techniques to object-oriented programming.

It should be noted that it is possible to achieve the benefits of object-oriented programming within procedural programming but the programming process becomes complex and the resultant code is difficult to maintain.

Classes versus Objects

Classes are the smallest units of design while objects are essentially the smallest units of execution. The objects interact with each other to provide the required functionality while the classes define the state and behaviour of those objects. In other words, an object is an **instance** of a class or a class defines an object's **type**.

Consequently, a class is a type. This is similar to the types, **int** and **char**. However, these are **primitive** types since they are built into the programming language and so cannot be changed. For example, a variable defined as type **int**, inherits the integer behaviour of the Java language. Therefore, integer operations, such as addition, + and subtraction, -, can be applied. Also, being integers, no decimal values are stored i.e., the data structure. In other words, the behaviour and data structure are predefined within the syntax and semantics of the Java programming language.

With classes, a programmer can define the behaviour and the structure of the data used, making them more advanced than primitive types. To use a class, you must declare a variable of the type of class in question, just as you would declare one of type **int**. Variables of a particular class type are called objects. Syntactically, to create and use a Java object, you must declare its **type** or class.

This is accomplished in two steps:

1. Declare the object
2. Use the **new** operator to create the instance.

Note that unlike C++, the declaration of an object does not create a usable instance of the class. Instead, it creates a reference and so the **new** operator must be used to access the functionality of the class. This feature is necessary so that many of the advanced Java object-oriented design techniques can be achieved (these features will be discussed in later handouts).

Consider the following example where a class of type `Player`, is created. For now, `Player` will be an empty class. The `VRWorldMain` class will be used to test the class.

```
public class Player
{
} // Player

// Actual execution begins in the function main of this class.
public class VRWorldMain
{
    public static void main( String args[] )
    {
        ➡      Player p1, p2;      // declare these objects as type Player

        ➡      p1 = new Player();  // create an instance of Player
    }
} // VRWorldMain
```

Both objects, `p1` and `p2`, have been initially declared as class type `Player`. In order to use `p1`, an instance of `Player` was created via the **new** operator (see figure 3). Note that `p2` will remain inaccessible until an instance of `Player` is created i.e. `p2 = new Player()`.

The **main** function is the starting point of all Java programs and so when a Java program is executed, the Java Virtual Machine (the Java interpreter) begins execution of the program at this point. As a result, the function prototype, for **main**, has been standardized as follows.

```
public static void main( String args[] )
{
} // main
```

The method declaration above must be followed when used in any Java class.

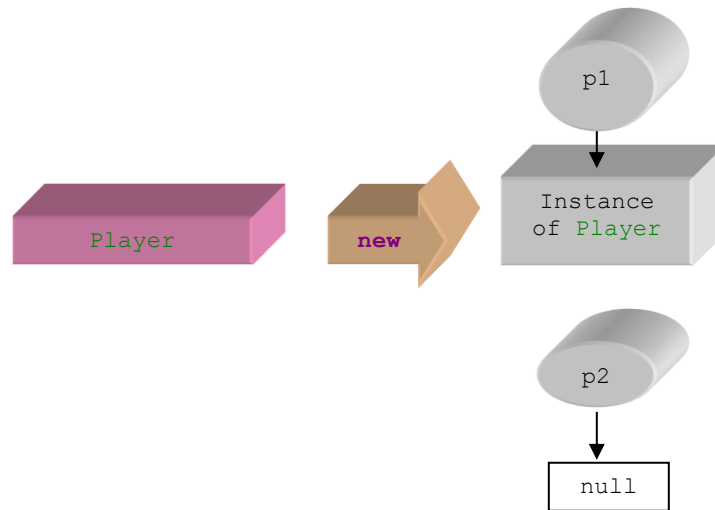


Figure 3

The Structure of a Class

A class defines the possible state and behaviour of any object that adopts that class as its type. In other words, the class defines the behaviour while the object is the actual use of that behaviour. As a result of this separation of definition and use, a program can have more than one object of a particular class type. For example, a car has four wheel objects i.e. each wheel object belongs to the class of wheels.

A class has two main sections:

- **Fields:** **Fields define the possible states.** The class defines the type and number of fields that an instance of it will contain. The fields hold the data for the instance of the class created.
- **Methods:** **Methods describe the allowed behaviour.** They state the actions that an instance of the class can perform. This is also known as the protocol of the class. Like the fields, specific methods can be hidden within the class while others are exposed (see the *Encapsulation* handout). Some texts use the term functions as opposed to methods.

The Structure of an Object

As stated previously, an object is an instance of a class i.e., the object belongs to a class. Remember, this separation is necessary so that more than one object can belong to a specific class. A newly created object is made up of the following four main sections:

- Id
- Type
- Fields

- Methods

The Id

The Id is the unique name assigned to an object, such as `p1` and `p2` in the previous example. It is a compulsory attribute since it is needed to distinguish one object from another.



Objects within the same scope cannot have the same Id.

The Type

When declared, each object is assigned its class type, for example `Player`. This attribute is also compulsory since it is required to ensure type safety and correct interface usage. When a programming language uses type safety it ensures:

- The class contains the named methods.
- The correct parameter types are passed to methods (functions).
- The types used in assignments are compatible. For example, an integer value can be assigned to a variable of type `int` but not one of type `char`.

Fields

Fields define the state of an object. Each instance of the class (each object) will receive its own copy of the fields, providing each object with its own unique state. These are also referred to as instance variables since a copy is created for each instance of a class. Consider the following example:

```
public class Player
{
    // Fields
    int age, height; // fields or instance variables
} // Player

public class VRWorldnMain
{
    public static void main( String args[] )
    {
        Player p1, p2;

        p1 = new Player();
        p2 = new Player();

        p1.age = 35;
        p1.height = 150;

        p2.age = 25;
        p2.height = 200;
    }
} // VRWorldMain
```

The class `Player` defines 2 fields, `age` and `height`. Any instance of `Player` will be given its own copy of the fields. In the previous example, both objects are of type `Player` but each contains its own values for `age` and `height` i.e. `p1 (35, 150)` and `p2 (25, 200)` (see figure 4).

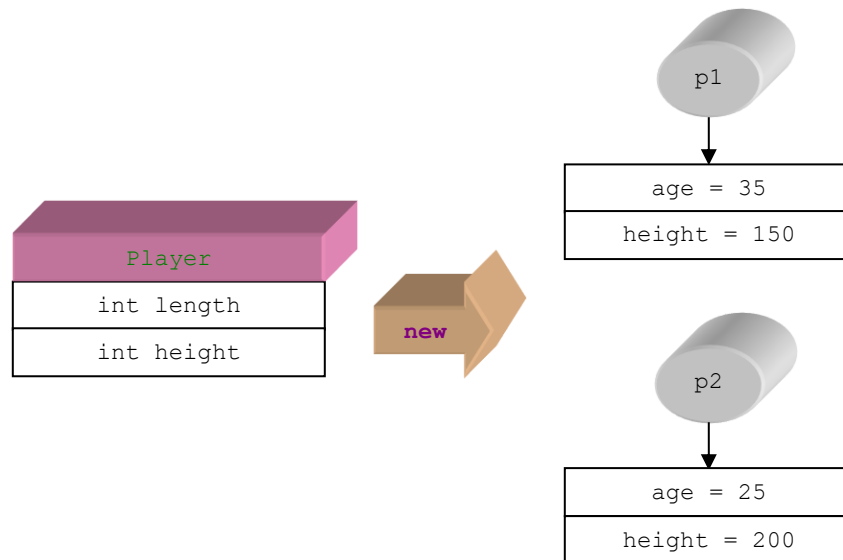


Figure 4



Fields are generally not exposed to the outside world (as in the previous example) and are hidden within the object. This will be discussed further in the *Encapsulation* handout.

Methods

Methods describe the behaviour of an object. All instances of a class share the same copy of the methods since they have the same behaviour. In our previous example, the fields for both `p1` and `p2` were accessed directly. By adding a method to set the field values, this direct access is no longer necessary. Consider the following example.

```

public class Player
{
    // Fields
    int age, height;

    // Methods

    void set( int newAge, int newHeight )
    {
        age = newAge;
        height = newHeight;
    } // set
} // Player

public class VRWorldMain
{
    public static void main( String args[] )
    {

```



```

    Player p1, p2;

    p1 = new Player();
    p2 = new Player();

    p1.set( 35, 150 );

    p2.set( 25, 200 );
}
} // VRWorldMain

```

A method set has now been added to the class `Player`. In this example, `p1` and `p2` use the same copy of the method set (see figure 5). This makes sense since there is no need to store more than one copy of the methods.

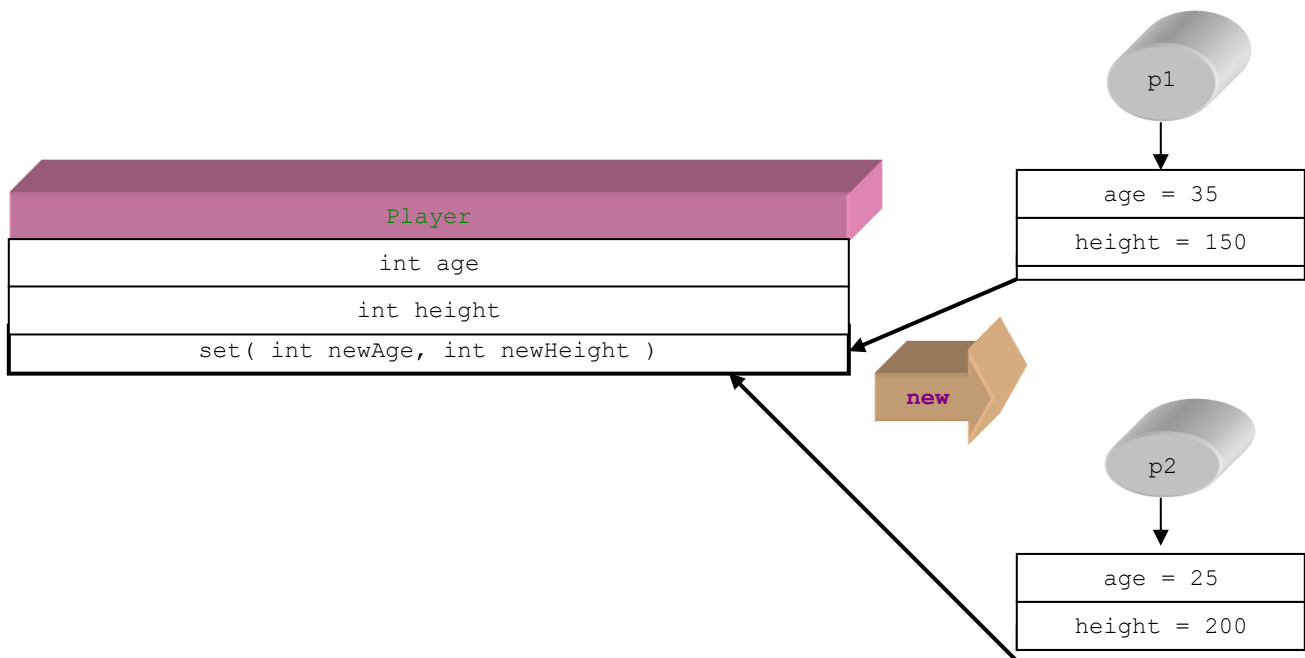


Figure 5

The new Operator

As stated previously, the `new` operator is used to create a usable instance of a class. When this operator is used, the **Java Virtual Machine (JVM)** allocates enough memory to hold all of the object's data. This raises the question of what happens to the memory when the object is no longer in scope. In C/C++, the user would have to manually de-allocate the memory using the `delete` operator. In Java this is performed automatically and is called *Garbage Collection*. Garbage Collection has the following advantages and disadvantages.

Advantages

- Reduces the possibility of memory leaks since memory is freed/recovered as soon as it is no longer in use. A memory leak occurs when unused memory is not released, resulting in the eventual consumption of all the available memory.
- An object that is being referenced cannot be deleted. This occurs mainly with pointers where it is possible to delete an object that is still being referenced by other pointers, resulting in some form of memory corruption.

Disadvantages

- Garbage collection can be one of the greatest bottlenecks in the speed of execution of a program, if not implemented correctly. Each time the garbage collector is activated, some aspects of the interpreter activity may be suspended. For example, if a call to the **new** operator is made and there is not enough memory, the execution of the code maybe suspended until the garbage collector frees enough memory.
- Makes the low-level aspect of systems programming very difficult since the level of control of the memory allocation/de-allocation process has been reduced.

The instanceof Keyword

Java provides the **instanceof** keyword for testing the type of an object. Consider the following code.

```
public class TypeTest
{
    public static void main( String args[] )
    {
        Player p = new Player();

        if( p instanceof Player )
        {
            System.out.println( "This is an instance of
                                Player." );
        }
        else
        {
            System.out.println( "This is NOT an instance of
                                Player." );
        }
    } // main
} // TypeTest
```

The output of this program will be,

```
This is an instance of Player.
```

Since `p` is of type `Player` then the test will return **true**.

Pointers

Java has no pointers but instead uses *references*. A reference is created when an object is declared. It provides access only to the functionality of the object and so the programmer does not have direct access to the memory, removing the dangers of pointer manipulation. For example:

- Pointers remove the concept of private data since they can provide direct access to that data.
- In some cases, integer values can be converted to pointer addresses which can result in incorrect addresses being formed and hence memory corruption.
- Arrays can be accessed directly by pointers. This can result in the program writing past the end of the array.

By using references, an assignment operation between two object-references in Java does not result in a copy of the values from one object to other. Instead, both object-references now refer to the same object. Consider the following example.

```
public class VRWorldMain
{
    public static void main( String args[] )
    {
        Player p1, p2;      // declare these objects as type
                           // Player i.e. the references

        p1 = new Player();

        p1.set( 35, 150 );

        ➡      p2 = p1;
    }
} // VRWorldMain
```

The references `p1` and `p2`, both refer to the same object.

For each object created, Java keeps a reference count. This is an integer value that states how many references are currently referring to the object. When this count becomes 0, the object can be removed from memory by the Java Garbage Collector.

Global Variables

In Java, the global name space is the class hierarchy and so one cannot create a variable outside of a class. This removes the possibility of side effects occurring on a system-wide basis due to some change in the state of a global variable. For example, it is impossible to guarantee that a global variable will be changed in a consistent and controlled manner. Java does allow a modified form of the global variable called static variables, which is discussed in the *Encapsulation* handout.