CHAPTER 3
Data Types

The most basic operation that a program in any programming language performs is manipulation of data. In order to achieve this, every programming language provides different types of data, operators and the syntax for valid statements. In this chapter, we discuss the various data types supported by the Java programming language.

All the data types in Java have been categorized into two—*primitive* data types and *reference* data types.

The primitive types, also known as built-in types, are `byte`, `short`, `int`, `long`, `float`, `double`, `char` and `boolean`.

The reference data types are *arrays*, *classes*, *interfaces*, *enums* and *annotations*. enums and annotations are introduced into the Java programming language from Java 5.

3.1 PRIMITIVE DATA TYPES

The primitive types may be further categorized into numeric types and `boolean` type. Table 3.1 lists the primitive data types.

3.1.1 `boolean` Data Type

A variable of the `boolean` type can have only two possible values, `true` or `false`. In Java, `boolean` is a separate data type and is not like C language, where a numeric value of non-zero or zero is considered as true or false, i.e. a numeric type cannot be used as `boolean`. E.g. we cannot use a numeric expression in the condition of the `if` statement, as is allowed in C, where `boolean` is not a separate data type.

3.1.2 Numeric Data Types

In the Java programming language, we do not have the `signed` or the `unsigned` prefix for the numeric data types. Note that signed and unsigned are not keywords in Java.

Among the numeric types, `byte`, `short`, `int` and `long` are the signed integral types, `char` is an unsigned integral type and the `float` and `double` are single-precision and double-precision signed floating-point data types.

TABLE 3.1 Primitive data types

`byte`	1 byte, signed, integral value in 2's complement
`short`	2 bytes, signed, integral value in 2's complement
`int`	4 bytes, signed, integral value in 2's complement
`long`	8 bytes, signed, integral value in 2's complement
`float`	Signed, single-precision floating-point value (4 bytes)
`double`	Signed, double-precision floating-point value (8 bytes)
`char`	2 byte, unsigned, integral value
`boolean`	Can have value of either `true` or `false` only (1 byte)

Consider the decimal value 2309737967 which can be written in hex as 89ABCDEF

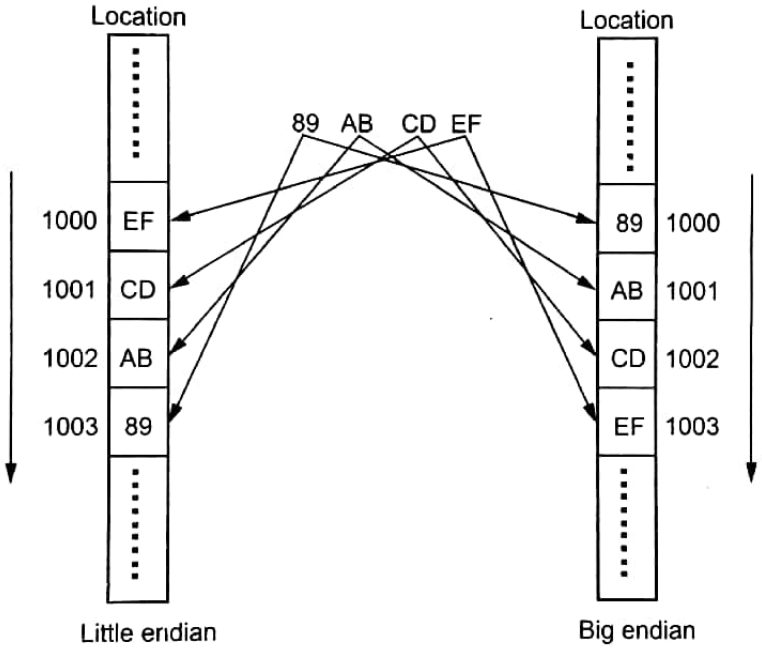


Figure 3.1 Byte ordering

3.1.2.1 Byte order

For all the numeric types whose size is more than 1 byte, there are two ways of ordering the bytes. The byte order for a machine could be big-endian or little-endian. The byte ordering is normally dependent on the architecture of the machine. All of the ×86 and ×64 machines follow the little-endian byte ordering for the numeric types, and most of the other machines, with RISC-based architecture follow the big-endian byte ordering. The Java Virtual Machine follows the big-endian byte ordering. The difference in the 2-byte orderings is that in case of the little-endian byte ordering, the most significant byte is at a higher location (comes at the end) and the least significant byte is at a lower location (comes first), whereas in case of the big-endian byte ordering, the most significant byte is at a lower location (comes first) and the least significant byte is at a higher location (comes at the end). Figure 3.1 shows the difference in byte ordering.

3.1.2.2 Integral data types

What is the size of `int` in C? Is it 2 bytes or 4 bytes? It depends on the platform. Most of the C compilers on the ×86 platform consider `int` as 2 bytes, whereas on the RISC machines, it

TABLE 3.2 Integral types and their value ranges

Data type	Minimum value	Maximum value
byte	-2^7 (−128)	$2^7 - 1$ (127)
short	-2^{15} (−32768)	$2^{15} - 1$ (32767)
char	0	$2^{16} - 1$ (65535)
int	-2^{31} (−2148473648)	$2^{31} - 1$ (2148473647)
long	-2^{63} (−9223372036854775808)	$2^{63} - 1$ (9223372036854775807)

TABLE 3.3 Floating-point types and their value ranges

Data type	Min. non-zero +ve value	Max. finite +ve value
float	1.401298464324817e-45f	3.4028234663852886e38f
double	4.9e-324	1.7976931348623157e308

is commonly found to be using 4 bytes, in some cases it may even use 8 bytes. Java works on single platform, the JVM; so on this platform the sizes of each of the data types are fixed. The size of byte is 1 byte, short is 2 bytes, int is 4 bytes and long is 8 bytes. The integral values are stored using 2's complement. The range of values for the various integral data types is given in Table 3.2.

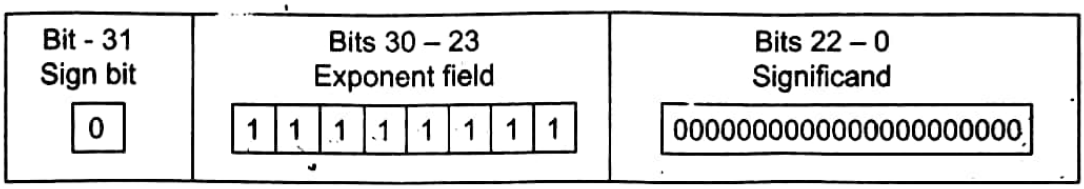
3.1.2.3 Floating-point data types

The types float and double represent the single-precision and double-precision floating-point values and their sizes are 4 and 8 bytes, respectively. These floating-point data type representations are according to the IEEE-754 standard for single-precision (32-bit) and double-precision (64-bit) floating-point values. This standard uses the most significant bit (MSB) as the sign bit, and the rest of the bits are divided among two fields, an *exponent* and a *significand*. In case of the single-precision value (float of Java), bit-31 is the sign bit, bits 30–23 are used for the exponent and bits 22–0 are used for the significand. In case of the double-precision value (double of Java), bit-63 is the sign bit, bits 62–52 are used for the exponent and bits 51–0 are used for the significand. The range of values for the two floating-point data types is given in Table 3.3.

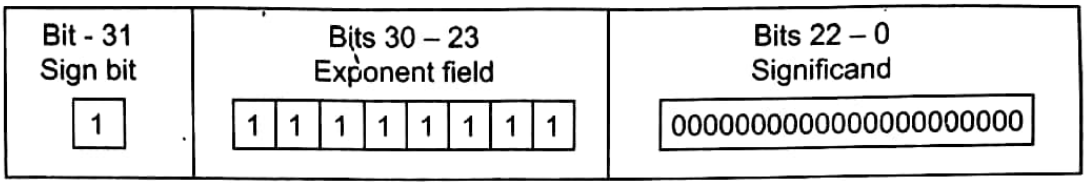
3.1.2.4 Infinities and NaNs for floating-point types

According to the IEEE 754, the floating-point numbers have a representation for positive infinity and negative infinity. There are also representations for the values that are Not-a-Number (NaN).

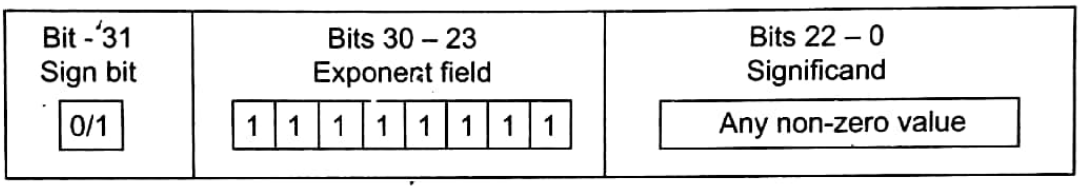
When all bits for the exponent field are one, then these numbers are used to represent the infinities and the NaN values. Figure 3.2 shows the bit representations for the infinities and the NaN values. When all exponent bits are one and all significand bits are zero, they represent infinity. Depending on the sign bit, the infinity is either a positive infinity or a negative infinity. When all exponent bits are one and the significand is a non-zero value, then they represent NaN values.



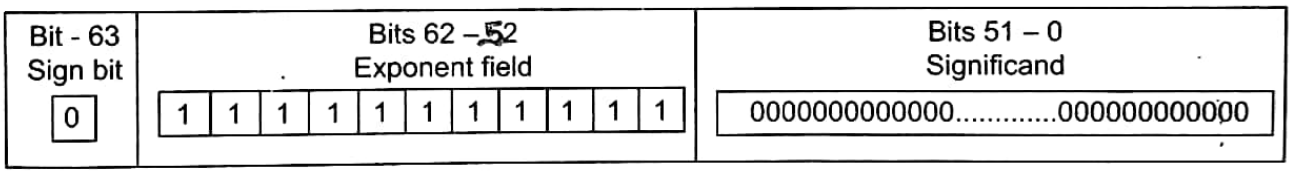
(a) Positive infinity for `float`



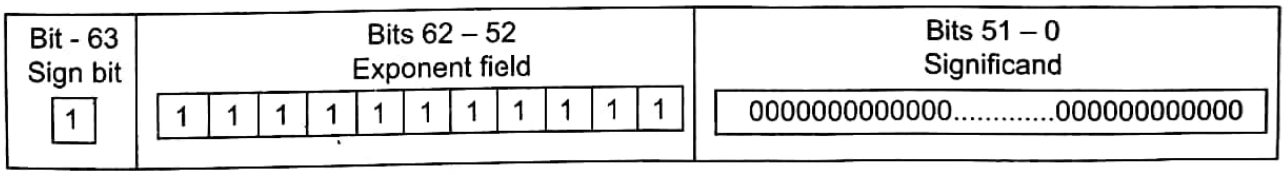
(b) Negative infinity for `float`



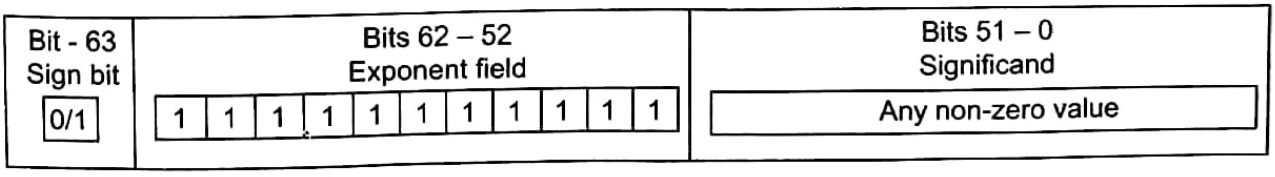
(c) NaN for `float`



(d) Positive infinity for `double`



(e) Negative infinity for ~~`float`~~ double



(f) NaN for `double`

Figure 3.2 Infinities and NaNs for floating-point data types

3.1.2.5 char data type

What is the size of char in C? The size of char is usually 1 byte in C. The size of char in Java is 2 bytes. It is an unsigned integral, 16-bit value, used for representing UTF-16 code-units.

Why is the size of a char, 2 bytes in Java? In C, char represents a character from the platform's local character set, which in most cases, is some extension of ASCII. The number of characters in most of these character sets is normally upto 256, so they require only 1 byte. In the case of Java, the char type is used to represent characters from the unicode character set using the UTF-16 encoding, which requires 16 bits. The details of the UTF encodings are given in Appendix A.

3.1.2.6 Unicode

Let us understand unicode. Unicode is a character set that has characters from all the languages of the world. There are various versions of the unicode character set. At the time of writing this book, the version of unicode was *5.2.0*. The unicode standard maps characters from all the langauges to a unique codepoint value. The codepoint values can be in the range of 0–10FFFF (Hex). This codepoint range has been divided into 17 planes, each of 65536 values, i.e. 2^{16}. The zeroeth plane, i.e. values from 0–FFFF(Hex) is known as BMP (Basic Multilingual Plane), and other planes define the supplementary characters. To represent the complete range of characters using only 16-bit units, the unicode standard defines an encoding called UTF-16. In this encoding, supplementary characters are represented as a pair of 16-bit code units, the first code unit from the high-surrogates range (D800 – DBFF(Hex)), and the second code unit from the low-surrogates range (DC00 – DFFF(Hex)). In unicode standard, the range of codepoint values from D800 to DFFF (Hex) has not been assigned to any valid character and is reserved for surrogates. For characters in the range of 0000–FFFF(Hex), the values of codepoints and UTF-16 code units are the same. The Java programming language represents text in sequences of 16-bit code units using the UTF-16 encoding. The char type in the Java programming language represents the 16-bit code unit.

3.1.3 Specifying Constants

How do we specify constants for the integral data types in Java? The JVM is a 32-bit machine. The basic integral type is int, i.e. even if we use operations on byte it would internally still be using 32 bits. How do we specify constants of type int? Let's look at the boolean expressions given in Listing 3.1.

Listing 3.1. Boolean expressions

```
1       (12 == 012)
2       (12 == 014)
```

Which of the above two boolean expressions are true? The first or the second? It seems that the first is true and the second is false; no, it is the other way round. The first expression is false and the second is true, not only in Java but also in C. In fact the token 019 is not a valid literal. Let us look at another expression.

```
(12 == 0x0C)
```

Will this expression evaluate to `true`? Yes, it will be `true`. When any token starts with '`0x`' or '`0X`', then the following characters are considered to be in hexadecimal digits. Similarly, when a token begins with a '`0`', then the next characters are considered as octal digits. So in the case of Java, an integer constant value can be specified either using octal, decimal or hexadecimal. To specify an `int` literal in hexadecimal, begin the literal with either '`0x`' or '`0X`', followed by the hexadecimal digits for the literal value. To specify an `int` literal in octal, begin the literal with a '`0`', followed by the octal digits for the literal value, and to specify an `int` literal in decimal, specify the literal value without starting with a zero.

If a literal of the long data type is required, then it must be suffixed by either 'l' (lowercase letter L) or 'L'. 'L' is preferred as we tend to misread 'l' (lowercase letter L) as the digit '1' (one).

Literals of the floating-point data types are also possible. When we use `7.5`, it is considered as literal of type `double`. If we want to have a literal of type `float`, it should be suffixed with the letter 'f' or 'F'.

Literals for the `char` types are declared by having the single character within two single quote characters. e.g.

```
'A' '0' '*'
```

For some special characters like the single quote character, double quote character, the new line character or tab character, we use the backslash character as an escape character and then specify these characters similar to C, e.g.

```
'\b' Backspace
'\f' Form feed
'\n' Line feed
'\r' Carriage return
'\t' Tab character
'\\' Back slash
'\'' Single quote
'\"' Double quote
```

`char` constants may also be specified using their octal values as in C, e.g.

```
'\017' character with octal value 017
```

3.2 UNICODE ESCAPES IN JAVA SOURCE CODE

The Java source code is a sequence of unicode characters. The Java source code can contain characters from any language and not just characters from the ASCII character set. Most of the time the source code is encoded in some native character set, which is an extension of ASCII. Even in these cases the Java source code can include characters that are not part of the native character set. This is done by using the unicode escape. In the source code we can specify any UTF-16 code unit by specifying the value as \u followed by four hexadecimal digits.

What are the rules for defining an identifier in Java? In Java, an identifier may contain any number of Java letters or Java digits, and it can start only with a Java letter. The sequence of Java letters in an identifier cannot match any of the keywords of the Java language or the `boolean` literals `true`, `false` or the literal `null`. A Java letter is not just the letters A–Z and a–z from

the ASCII character set, but it also includes the letters from other languages available from the unicode character set. The Java letter also includes the connecting punctuation characters like the '_' character, currency symbols like the '$' sign or a numeric letter like the roman numeral'. The Java letter or digit also includes the digits used in the various languages available in the unicode character set and not just the digits 0–9 from the ASCII character set. It also includes the combining marks and the non-spacing marks, which may be used for combining characters, e.g. we can use the null character with the value of zero as a combining mark between two letters. The following declaration shows a valid declaration of a Java identifier:

```
char અ = 'અ';
```

Here અ has been used as an identifier; since it is a letter in Gujarati, this declaration is valid. But then how do we use these characters in a Java file, which is created using a text editor, where only the ASCII characters are available? In a Java file before the compiler identifies the lines and the tokens, it looks for unicode escapes in the Java file. The Java compiler works on unicode characters. Our Java source file is normally encoded in ASCII or some extension of ASCII. While decoding from ASCII to unicode, the compiler would first replace the unicode escapes in the Java file with the actual unicode character value. Using the unicode escape we can write the above declaration in a Java source file encoded in ASCII as shown below:

```
char \u0A85 = '\u0A85'; //0A85 is the hex value for અ
```

Unicode escape is written as \u followed by four hexadecimal digits, where the hexadecimal digits are the codepoint values for that character in the unicode character set.

The following code segment would not compile:

```
char ch = '\u000A';
```

since this will be seen by the Java compiler as:

```
char ch = '
```

Instead `char ch ='\n';` would do the thing for us here.

3.3 REFERENCE DATA TYPES

Java follows the object reference model, which is followed by almost all Object-oriented Programming Languages. C++ differs from Java mainly because it does not follow the Object Reference Model. Java does not use pointers, instead it uses references. References are somewhat like pointers in C, but they are different from pointers in the sense that unlike pointers which can point to any memory location, references can only refer to objects (instances of a reference type). The reference variables are declared to refer to instances of a particular type, and can only refer to instances of that type. They can never be made to refer to instances of any other type, which is always possible with pointers, and can be a source of lots of errors.

Let us consider the following declarations:

```
Rectangle r;    Date d;
```

Here r and d are variables of type Rectangle and Date classes, respectively. How much space is allocated for the variable r or for variable d in the above declarations? Let's assume that the class Date has three integers to store the date, month and the year parts; and the class Rectangle has two integers to store the length and the width. In the above declaration the size allocated for both r and d would be the same. The r and d above are not the objects of type Rectangle or Date, they are simply references that can refer to an instance of Rectangle and Date, respectively. In the above declaration the space is only allocated for a reference. For all reference variables the space allocated is the same. What is guaranteed in Java is that r will not be able to refer to an object which is not a Rectangle, and d will not be able to refer to an instance that is not a Date. The difference between reference and pointer is that a pointer is an address. We can manipulate it the way we like. It could be pointing anywhere, and we can manipulate its value to refer to any location. Therefore, we can make a pointer of Rectangle point to an instance of Date or vice-versa. This could lead to errors that many times are difficult to resolve.

How do we create an instance. To create an instance we use the new operator for allocation purpose. The instance of Date is allocated using the new operator as shown:

```
d = new Date();
```

In the above code, d is a reference variable that refers to an instance of Date class. Here d is not an instance, it is simply a reference to an instance. Also d may seem to look like a pointer. We operate on instances using a reference. The references allow us to operate on the instance referred by the reference. We can use d to access the instance referred by d. This variable d is different from a pointer because we don't manipulate references the way we do with pointers in other programming languages. There are restrictions on how reference variables are assigned values. We cannot assign a memory location to a reference variable, the way we can for pointers. Whatever is the type declared for a reference variable, it can never be made to refer to the instance, which is not of that type. It is not an address that can be made to point anywhere.

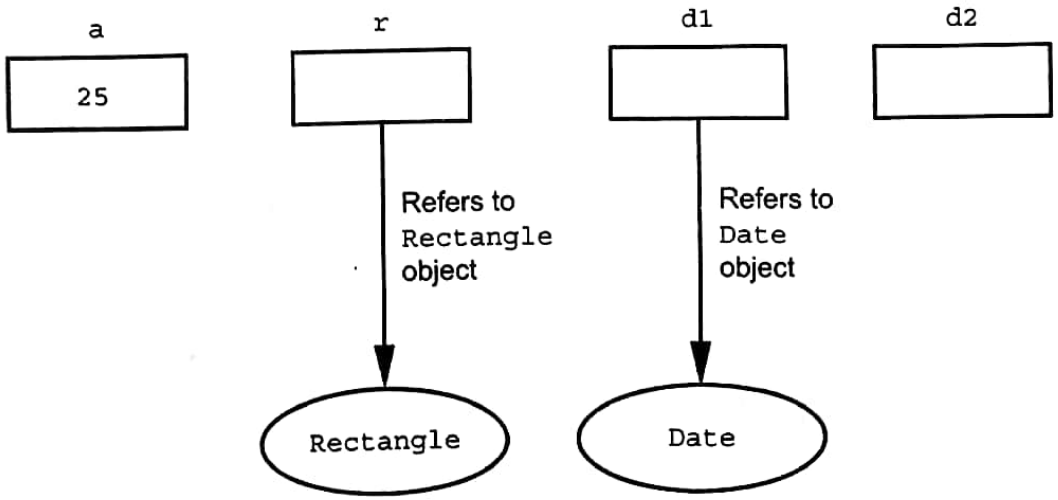


Figure 3.3 Reference variables refer to objects

A major difference of Java with other programming languages is that in case of Java, the developer does not have any mechanism to directly deallocate any instance that has been allocated. The work of deallocating any instance is done by the JVM itself. The JVM would decide about which instances are currently not useable (there are no reference variables referring to an instance), and then would deallocate such instances. This work of identifying and deallocating instances is done by the garbage collector in the JVM.

Using the reference variables, it is possible for a single instance to be referred by more than one reference variables as given in Listing 3.2.

Listing 3.2. Multiple references to a single instance

```
1    Date d1, d2;
2    d1 = new Date();
3    d2 = d1;
4    ...
```

In Line 1, two reference variables `d1` and `d2` are declared of type `Date`, in Line 2, `d1` starts referring to the instance of `Date` allocated using the `new` operator. Now in Line 3, `d2` is also made to refer to the same instance as `d1` (the one which was allocated in Line 2). So here we have `d1` and `d2` referring to the same instance. Therefore, after Line 3 in the code of Listing 3.2, any access to the object of `Date` class done using variable `d1` or `d2` operates on the same instance of `Date`.

There is also a literal for the reference data type called `null`. This is a constant that is assignable to any reference variable. Whenever any reference variable is assigned a `null`, then that reference variable is not referring anywhere. Therefore, in the earlier listing after Line 3, if we add another line as "`d1=`**null**;", then this line would result in `d1` to stop referring to the instance created in Line 2. If we update the code as given in Listing 3.3, then this would result in the instance created in Line 2 not being referred by any reference variable in Line 5. So after this Line 5, the instance created in Line 2 becomes available for garbage collection, and may be deallocated by the garbage collector within the JVM. After Line 5, any attempt to access an instance of `Date` using `d1` or `d2` would result in an error at runtime.

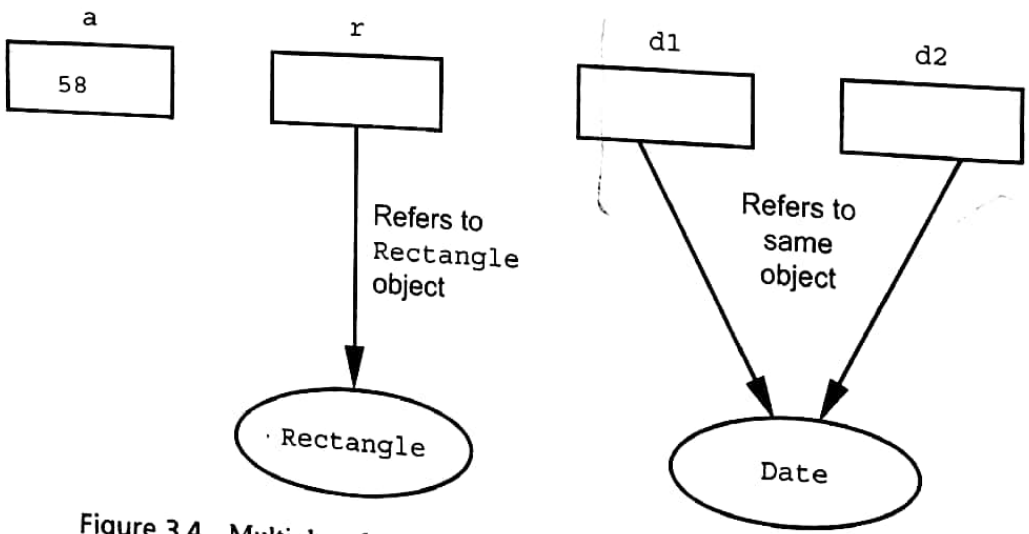


Figure 3.4 Multiple reference variables, referring to the same instance

Listing 3.3. Losing reference to instance

```
1    Date d1, d2;
2    d1 = new Date();
3    d2 = d1;
4    d1 = null;
5    d2 = d1;
6    ...
```

3.3.1 Arrays

Let us look at arrays as reference types. How do we create arrays in C? 'int a[size]'. Here, we declare and allocate at the same time. This type of declaration is not valid in Java. We break it into two steps for Java.

Step 1:

```
int intArray[];
```

This is simply a declaration of a reference variable called intArray, which can refer to an instance of type int[]. The size is not specified. intArray is the variable of type int[], a more logical way to make the above declaration is given below:

```
int[] intArray;
```

Here, `int[]` is the data type and `intArray` is the variable. It is a reference that can refer to any instance of `int[]`. Both the above ways of declaration of `intArray` are equivalent.

How do we create an instance of an array?

Arrays are instances that have a fixed number of elements of a uniform type. Each element in the array could be accessed using its index position. An instance of the array type can be created by specifying the type and the number of elements to the new operator as given below:

```
new <type>[<size>]
```

where `<type>` can be any data type and `<size>` is any integer expression, e.g. `new int[10];`

We may declare references of array types as given below:

```
1  <type>[] <variable-name>  or
2  <type> <variable-name>[]
3  eg. int[] intArray   or int intArray[]
```

Both the forms of declarations given above are equivalent. In the above code, `intArray` is only a reference and not an instance of an array. In the above example, the reference `intArray`, which is declared to be type `int[]` (array of `int` type), can refer to an instance of the array of `int` only and not to any other type.

We can initialize a reference using the assignment as follows:

```
1  int[] intArray;
2  intArray = new int[10];
```

In the first statement intArray is declared as a reference variable and in the second line an instance of **int**[] with 10 elements is allocated using the new operator, and intArray starts refering to that instance. An element in the array referred by intArray may be accessed using the array index operator [], e.g. to access the element in the array at index position 5 (index positions start from 0, i.e. the first element is at index position 0), we can use intArray[5]. Therefore, in the above case we have array elements with index positions from 0 to 9, and these elements can be accessed as intArray[<**int**-expression>] where <**int**-expression> evaluates to a value in the range of 0 – 9. In case we try to access elements by using the index position outside this range, then it results in an error at the runtime. It would not give us wrong values by accessing any memory address that has not been allocated (which happens in case of pointers and arrays in C). Each element of the intArray is of type int, so the expression intArray[5] is of type int.

The number of elements in an array can be obtained by using its length variable. Therefore, to know the number of elements in an array referred by an array reference variable intArray, we can use intArray.length. Whenever an instance of array is created, the number of elements is fixed and cannot be changed. Therefore, typically we use this length in loops where we want to process all the elements in an array, as shown in Listing 3.4.

Listing 3.4. Using length on array instances

```
1 int sum(int[] intArray) {
2     int sum = 0;
3     for (int index = 0; index < intArray.length; index++) {
4         sum += intArray[index];
5     }
6     return sum;
7 }
```

The initial value for the elements of an array of numeric type allocated using the new operator will be 0. The initial value for elements in a boolean array will be false.

3.3.2 Array of Reference Type

Arrays can be created for any data type, including the reference data types. Say for example, we want an array of String with 10 elements. It may be declared and initialized as given below:

```
String[] strArray = new String[10];
```

This would allocate an array with 10 elements. The elements in this array are not the String objects. They are references to a String. The initial value for these elements will be null.

We could initialize the elements in this array using the assignment as given below:

```
strArray[0] = "Hello";
strArray[1] = "world";
...
```

Figure 3.5 shows the references initialized using the above statements.

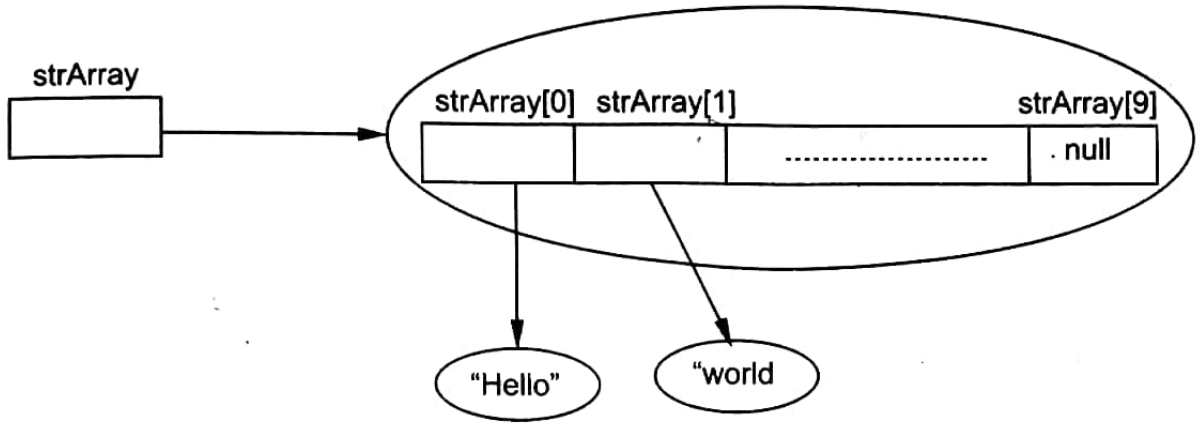


Figure 3.5 An array of string

3.3.3 Two-dimensional and Multi-dimensional Arrays

What is a two-dimensional array? In Java, a two-dimensional array is an array of array type. Therefore, a two-dimensional array of `int` is an array of `int[]`, and array is a reference type. So, to declare a two-dimensional array of `int`, we can declare in either of the following ways:

```
int twoDimArray[][]; int[] twoDimArray[]; int[][] twoDimArray;
```

All the three declarations given above are equivalent. The last declaration is preferred. There are two ways of allocating a two-dimensional array, using the `new` operator, as shown in the example given below:

```
twoDimArray = new int[10][5];
twoDimArray = new int[10][];
```

When allocating a two-dimensional array using the `new` operator, the size of the first dimension is mandatory, and the size of the second dimension is optional.

The difference between the two allocations is as shown in Figure 3.6.

3.3.4 Classes

The classes are the reference data types that follow a strict hierarchial relationship of super-class and sub-class, with the class called `java.lang.Object` as the root of the class hierarchy. A class cannot have more than one direct super-class. A class may implement any number of interfaces. There are a lot of classes, which are available as part of JDK. A developer also defines his or her own classes as per his or her requirements. Most of the members of a class definition are discussed in Chapter 5, and the nested member types of classes and interfaces are further discussed in Chapter 12.

3.3.5 Interfaces

The interfaces are the reference data types whose instances are not created and which do not follow a strict hierarchy, i.e. an interface can have any number of super-interfaces. The various kinds of members for an interface are discussed in detail in Chapter 6, and the nested member types in an interface are further discussed in Chapter 12.

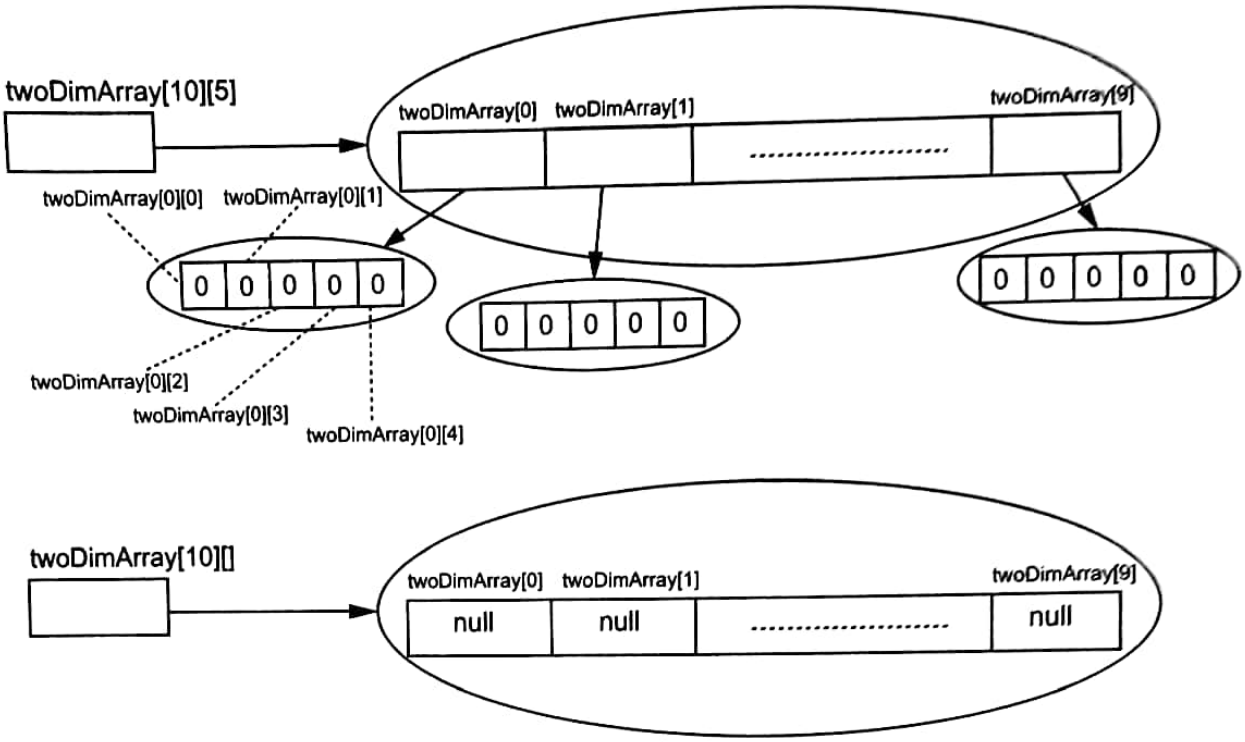


Figure 3.6 Two-dimensional array allocation

3.3.6 Enum

The enum types were introduced in Java programming from Java Version 5.0. These are special classes that have a fixed number of instances. All these instances are known at the compile time. The instances of the enum types can be fetched either by their name or by their ordinal values. These are discussed in detail in Chapter 12.

3.3.7 Annotation

Annotations were introduced in Java programming from Java Version 5.0. These are special kinds of interfaces that are used to annotate programming elements in the source code. These could then be used by annotation processing tools. Annotations have been discussed in Chapter 23.

3.4 SUPER-TYPES AND SUB-TYPES

Let us look at relationships between data types. We have a relationship of super-type and sub-type between data types. A data type is a sub-type of a given data type if it is a special kind of the given data type. The sub-type would then always be usable wherever the super-type is usable, e.g. in case of primitive types, `int` is a sub-type of `long`, as `int` can always be used wherever `long` is used. All ints are long, but not all longs may be ints, i.e. we cannot use the super-type for sub-type. In the Java programming language, for the primitive data types, the super-type and the sub-type relationships are as given in Figure 3.7.

For reference data types, the super-type sub-type relationships involve inheritance, which is explained in Chapters 6 and 7. The `java.lang.Object` class is the super-type for all the reference data types. There are a couple of special interfaces `java.lang.Cloneable` and `java.io.Serializable`, which are super-types for all the array types.

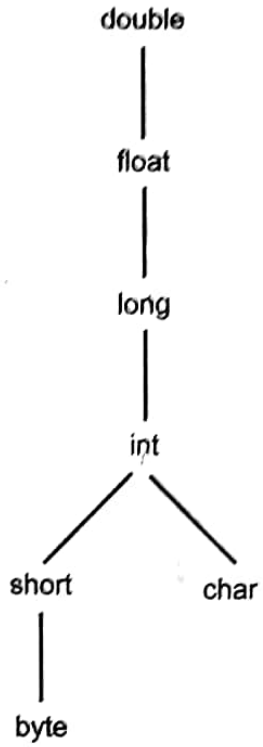


Figure 3.7 Super-type and sub-type relations in primitive data types

The super-types for any class, are

(a) The direct super-class of the class
(b) All the super-types of this super-class
(c) The interfaces directly implemented by the class

The super-types for any interface are

(a) The `java.lang.Object` class
(b) All the interfaces directly extended by the interface
(c) All the super-types of the interfaces directly extended by the interface

The super-types for an array of any primitive type are

(a) The `java.lang.Object` class
(b) The `java.lang.Cloneable` interface
(c) The `java.io.Serializable` interface

The super-types for an array of any reference type `referencetype` are

(a) The arrays of all the super-types of the reference type `referencetype`
(b) The class `java.lang.Object`
(c) The interfaces `java.lang.Cloneable` and `java.io.Serializable`

LESSONS LEARNED

- Java has mainly two kinds of data types—the primitive and the reference data types. The primitive data types are the `byte`, `short`, `int`, `long`, `float`, `double`, `char` and `boolean`. The reference data types are (a) arrays, (b) classes and (c) interfaces. *enum* and *annotations* are special kinds of classes and interfaces.
- `float` and `double` are the floating-point data types, which follow the IEEE-754 standard, which has representations for the infinities (result of a floating-point division by zero) and not-a-number.
- The variables of reference types are not instances themselves, but are references to instances.
- Multi-dimensional arrays are arrays of an array type
- The size of an array is fixed and can be retrieved using the length variable on its instance, e.g. if `a` refers to an array instance then `a.length` returns the number of elements in the array referred by `a`.
- Data types may have a super-type and a sub-type relationship.
- Identifiers in Java can contain Java letters or digits, a separator or a currency symbol, and cannot start with a digit. The identifiers cannot be keywords of the language or the literals `true`, `false` and `null`.
- Anywhere in a Java file, we can specify any kind of unicode character, which may not be available in the native character set by using a unicode escape. Unicode escapes are specified by using \followed by four hexadecimal digits for the unicode value of the character.

EXERCISES

1. State which of the following are true or false:

(a) The size of `char` datatype is 1 byte.

(b) Java uses ASCII character set of the `char` data type.

(c) The size of `int` in Java is platform dependent.

(d) The floating-point division by zero in Java would result in an error at runtime.

(e) Java uses the little-endian byte ordering.

(f) An array subscript should normally be of data type `float`.

(g) To indicate that 100 locations should be reserved for `int` array `p`, the programmer writes the declaration `int p[100];`.

(h) The size of `long` is 4 bytes.

(i) The number of elements in an array `x` is available as `x.size;`.

(j) In a two-dimensional array, each row can have different number of columns.

(k) Each reference variable always refers to a different instance, i.e. two reference variables cannot refer to the same instance.

(l) A single unit of java primitive type `char` can be used to represent any unicode codepoint.

(m) \ŏ905 is a valid identifier in Java (\ŏ905 is a letter in the Devanagari block of the unicode character set).

2. Fill in the blanks:
   (a) The default byte order in Java is _____.
   (b) The size of `long` data type in Java is _____ bytes.
   (c) The number of exponent bits in `float` data type is _____.
   (d) Java uses _____ standard for floating-point representation.
   (e) Java uses _____ character set to represent text data.
   (f) For numeric types Java follows _____ byte ordering.
   (g) The codepoint values for unicode can be in the range from _____ to _____ (specify values in hexadecimal).
   (h) _____ can be assigned to a reference variable to make the variable to stop referring to any instance.
3. Explain the difference between the variables of primitive data types and reference data types.
4. What are the two types of byte ordering? Explain the difference between them.
5. How do we specify integer constants? Specify the three different ways of specifying an integer constant.
6. Explain the use of unicode escapes.
7. Explain, with example, how an instance becomes eligible for garbage collection.
8. What will be the output in Line 8 if the following code segment is run?

```
1    int[][] a = new int[10][5];
2    a[2] = a[0];
3    for(int i=0; i<a.length; i++){
4        for(int j=0; j<a[i].length; j++){
5            a[i][j] = i+j;
6        }
7    }
8    System.out.println(a[0][3]);
```

   (a) 0
   (b) 3
   (c) 5
   (d) 8
   (e) None of the above
9. Which of the following are valid data types in Java?
   (a) Unsigned byte
   (b) Short
   (c) Long
   (d) Character
   (e) Boolean
   (f) java.lang.String