5 Some Standard Classes

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.
—John von Neumann (1951)

Learning Objectives

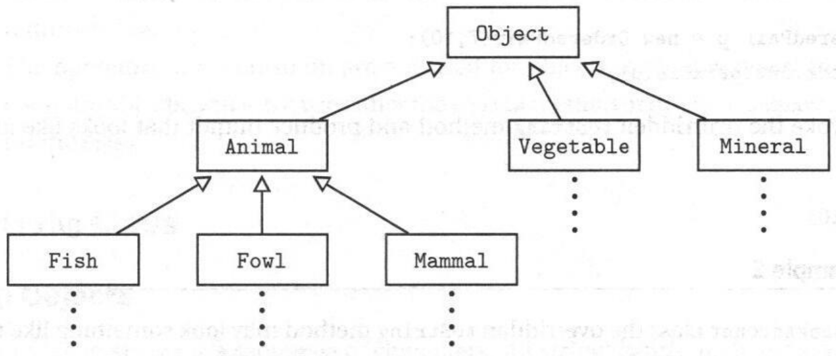
In this chapter, you will learn:

- ➔ The `Object` class
- ➔ The `String` class
- ➔ Wrapper classes
- ➔ The `Math` class

The `Object` Class

The Universal Superclass

Think of `Object` as the superclass of the universe. Every class automatically extends `Object`, which means that `Object` is a direct or indirect superclass of every other class. In a class hierarchy tree, `Object` is at the top:



Methods in `Object`

There are many methods in `Object`, all of them inherited by every other class. The expectation is that these methods will be overridden in any class where the default implementation is not suitable. The methods of `Object` in the AP Java subset are `toString` and `equals`.

The `toString` Method

```
public String toString()
```

This method returns a version of your object in `String` form.

When you attempt to print an object, the inherited default `toString` method is invoked, and what you will see is the class name followed by an `@` followed by a string of characters, which is not what you intended. For example,

```
SavingsAccount s = new SavingsAccount(500);
System.out.println(s);
```

produces something like

```
SavingsAccount@fea485c4
```

To have more meaningful output, you need to override the `toString` method for your own classes. Even if your final program doesn't need to output any objects, you should define a `toString` method for each class to help in debugging.

Example 1

```
public class OrderedPair
{
    private double x;
    private double y;

    //constructors and other methods ...

    /** Returns this OrderedPair in String form. */
    public String toString()
    {
        return "(" + x + "," + y + ")";
    }
}
```

Now the statements

```
OrderedPair p = new OrderedPair(7,10);
System.out.println(p);
```

will invoke the overridden `toString` method and produce output that looks like an ordered pair:

```
(7,10)
```

Example 2

For a `BankAccount` class the overridden `toString` method may look something like this:

```
/** Returns this BankAccount in String form. */
public String toString()
{
    return "Bank Account: balance = $" + balance;
}
```

The statements

```
BankAccount b = new BankAccount(600);
System.out.println(b);
```

will produce output that looks like this:

```
Bank Account: balance = $600
```

Note

1. The + sign is a concatenation operator for strings (see p. 174).
2. Array objects are unusual in that they do not have a `toString` method. To print the elements of an array, the array must be traversed and each element must explicitly be printed.

The `equals` Method

```
public boolean equals(Object other)
```

All classes inherit this method from the `Object` class. It returns `true` if this object and `other` are the same object, `false` otherwise. Being the same object means referencing the same memory slot. For example,

```
Date d1 = new Date("January", 14, 2001);
Date d2 = d1;
Date d3 = new Date("January", 14, 2001);
```

Do not use == to test objects for equality. Use the `equals` method.

The test `if (d1.equals(d2))` returns `true`, but the test `if (d1==d3)` returns `false`, since `d1` and `d3` do not refer to the same object. Often, as in this example, you may want two objects to be considered equal if their *contents* are the same. In that case, you have to override the `equals` method in your class to achieve this. Some of the standard classes described later in this chapter have overridden `equals` in this way. You will not be required to write code that overrides `equals` on the AP exam.

Note

1. The default implementation of `equals` is equivalent to the == relation for objects: In the `Date` example above, the test `if (d1 == d2)` returns `true`; the test `if (d1 == d3)` returns `false`.
2. The operators <, >, and so on are not used for objects (reference types) in Java. To compare objects, you must use either the `equals` method or define a `compareTo` method for the class.

The `String` Class

`String` Objects

An object of type `String` is a sequence of characters. All *string literals*, such as `"yikes!"`, are implemented as instances of this class. A string literal consists of zero or more characters, including escape sequences, surrounded by double quotes. (The quotes are not part of the `String` object.) Thus, each of the following is a valid string literal:

```
""                  //empty string
"2468"
"I must\n go home"
```

`String` objects are *immutable*, which means that there are no methods to change them after they've been constructed. You can, however, always create a new `String` that is derived from an existing `String`.

Constructing `String` Objects

A `String` object is unusual in that it can be initialized like a primitive type:

```
String s = "abc";
```

This is equivalent to

```
String s = new String("abc");
```

in the sense that in both cases `s` is a reference to a `String` object with contents `"abc"` (see Box on p. 176).

It is possible to reassign a `String` reference:

```
String s = "John";
s = "Harry";
```

This is equivalent to

```
String s = new String("John");
s = new String("Harry");
```

Notice that this is consistent with the immutable feature of `String` objects. `"John"` has not been changed; he has merely been discarded! The fickle reference `s` now refers to a new `String`, `"Harry"`. It is also OK to reassign `s` as follows:

```
s = s + " Windsor";
```

`s` now refers to the object `"Harry Windsor"`.

Here are other ways to initialize `String` objects:

```
String s1 = null;            //s1 is a null reference
String s2 = new String();    //s2 is an empty character sequence

String state = "Alaska";
String dessert = "baked " + state;  //dessert has value "baked Alaska"
```

The Concatenation Operator

The `dessert` declaration above uses the *concatenation operator*, `+`, which operates on `String` objects. Given two `String` operands `lhs` and `rhs`, `lhs + rhs` produces a single `String` consisting of `lhs` followed by `rhs`. If either `lhs` or `rhs` is an object other than a `String`, the `toString` method of the object is invoked, and `lhs` and `rhs` are concatenated as before. If one of the operands is a `String` and the other is a primitive type, then the non-`String` operand is converted to a `String`, and concatenation occurs as before. If neither `lhs` nor `rhs` is a `String` object, an error occurs. Here are some examples:

```
int five = 5;
String state = "Hawaii-";
String tvShow = state + five + "-0";  //tvShow has value
                                      //"Hawaii-5-0"
int x = 3, y = 4;
String sum = x + y;         //error: can't assign int 7 to String
```

Suppose a `Date` class has a `toString` method that outputs dates that look like this: `2/17/1948`.

```
Date d1 = new Date(8, 2, 1947);
Date d2 = new Date(2, 17, 1948);
String s = "My birthday is " + d2;  //s has value
                                    //"My birthday is 2/17/1948"
String s2 = d1 + d2;    //error: + not defined for objects
String s3 = d1.toString() + d2.toString();  //s3 has value
                                            //8/2/19472/17/1948
```

Comparison of String Objects

There are two ways to compare String objects:

1. Use the equals method that is inherited from the Object class and overridden to do the correct thing:

   ```
   if (string1.equals(string2)) ...
   ```

   This returns true if string1 and string2 are identical strings, false otherwise.
2. Use the compareTo method. The String class has a compareTo method:

   ```
   int compareTo(String otherString)
   ```

   It compares strings in dictionary (lexicographical) order:

   - If string1.compareTo(string2) < 0, then string1 precedes string2 in the dictionary.
   - If string1.compareTo(string2) > 0, then string1 follows string2 in the dictionary.
   - If string1.compareTo(string2) == 0, then string1 and string2 are identical. (This test is an alternative to string1.equals(string2).)

Be aware that Java is case-sensitive. Thus, if s1 is "cat" and s2 is "Cat", s1.equals(s2) will return false.

Characters are compared according to their position in the ASCII chart. All you need to know is that all digits precede all capital letters, which precede all lowercase letters. Thus "5" comes before "R", which comes before "a". Two strings are compared as follows: Start at the left end of each string and do a character-by-character comparison until you reach the first character in which the strings differ, the *k*th character, say. If the *k*th character of s1 comes before the *k*th character of s2, then s1 will come before s2, and vice versa. If the strings have identical characters, except that s1 terminates before s2, then s1 comes before s2. Here are some examples:

```
String s1 = "HOT", s2 = "HOTEL", s3 = "dog";
if (s1.compareTo(s2) < 0))        //true, s1 terminates first
    ...
if (s1.compareTo(s3) > 0))        //false, "H" comes before "d"
```

Don't Use == to Test Strings!

The expression `if (string1 == string2)` tests whether `string1` and `string2` are the same reference. It does not test the actual strings. Using == to compare strings may lead to unexpected results.

Example 1

```
String s = "oh no!";
String t = "oh no!";
if (s == t) ...
```

The test returns `true` even though it appears that `s` and `t` are different references. The reason is that, for efficiency, Java makes only one `String` object for equivalent string literals. This is safe in that a `String` cannot be altered.

Example 2

```
String s = "oh no!";
String t = new String("oh no!");
if (s == t) ...
```

The test returns `false` because use of `new` creates a new object, and `s` and `t` *are* different references in this example!

The moral of the story? Use `equals` not == to test strings. It always does the right thing.

Other String Methods

The Java `String` class provides many methods, only a small number of which are in the AP Java subset. In addition to the constructors, comparison methods, and concatenation operator + discussed so far, you should know the following methods:

```
int length()
```

Returns the length of this string.

```
String substring(int startIndex)
```

Returns a new string that is a substring of this string. The substring starts with the character at `startIndex` and extends to the end of the string. The first character is at index zero. The method throws an `IndexOutOfBoundsException` if `startIndex` is negative or larger than the length of the string. Note that if you're using Java 7 or above, you will see the error `StringIndexOutOfBoundsException`. However, the AP Java subset lists only `IndexOutOfBoundsException`, which is what they will use on the AP exam.

```
String substring(int startIndex, int endIndex)
```

Returns a new string that is a substring of this string. The substring starts at index `startIndex` and extends to the character at `endIndex-1`. (Think of it this way: `startIndex` is the first character that you want; `endIndex` is the first character that you *don't* want.) The method throws a `StringIndexOutOfBoundsException` if `startIndex` is negative, or `endIndex` is larger than the length of the string, or `startIndex` is larger than `endIndex`.

```
int indexOf(String str)
```

Returns the index of the first occurrence of `str` within this string. If `str` is not a substring of this string, `-1` is returned. The method throws a `NullPointerException` if `str` is null.

Here are some examples:

```
"unhappy".substring(2)         //returns "happy"
"cold".substring(4)            //returns "" (empty string)
"cold".substring(5)            //StringIndexOutOfBoundsException
"strawberry".substring(5,7)    //returns "be"
"crayfish".substring(4,8)      //returns "fish"
"crayfish".substring(4,9)      //StringIndexOutOfBoundsException
"crayfish".substring(5,4)      //StringIndexOutOfBoundsException
"crayfish".substring(4,"crayfish".length());  //returns "fish"
"crayfish".substring(4);        //returns "fish".
                                //This is a less error-prone form of the
                                // substring method if you want to start at
                                // an index and return the rest of the string.

String s = "funnyfarm";
int x = s.indexOf("farm");   //x has value 5
x = s.indexOf("farmer");     //x has value -1
int y = s.length();          //y has value 9
```

Special Emphasis

The `String` methods `substring` and `indexOf` are used frequently on the AP exam. Be sure that you recall:

- The first index of a `String` is `0`.
- The method call `s.substring(start, end)` returns the substring of `s` starting at index `start` but ending at index `end-1`.
- The method call `s.indexOf(sub)` returns the index of the first occurrence of substring `sub` in `s`.
- `s.indexOf(sub)` returns `-1` if `sub` is not in `s`.

You should be nimble and well practiced in processing strings.

Example 1

The following type of code can be used to look for multiple occurrences of a substring in a given string.

```
int pos = s.indexOf(someSubstring);
while (pos >= 0)                  //the substring was found
{
    doSomething();
    s = s.substring(pos + 1);  //throw away all characters of s
                                  //up to and including someSubstring

    pos = s.indexOf(someSubstring);  //Is there another occurrence
                                        //of someSubstring?

}
```

A modified version of the above code, using some combination of a loop, `indexOf`, and `substring`, can be used to

- count number of occurrences of `substring` in `str`.
- replace all occurrences of `substring` in `str` with `replacementStr`.
- remove all occurrences of `substring` in `str`.

Often conditionals are used to search for keywords that will trigger different responses from a program. Using `if` and `if...else` should be second nature to you.

Example 2

The user will enter a `sentence` and the program will produce a `Reply`.

```
if (sentence.indexOf ("love") != -1)
{
    if (sentence.indexOf ("you") != -1)
        Reply = "I'm in heaven!";
    else
        Reply = "But do you love me?";
}
else
   Reply = "My heart is in pieces on the floor.";
```

Here are some possible sentences that the user may enter, with the corresponding `Reply`:

Sentence	Reply
`I love chocolate cake.`	`But do you love me?`
`I love chocolate cake; do you?`	`I'm in heaven!`
`I hate fudge.`	`My heart is in pieces on the floor.`

If the substring `"love"` isn't in the sentence, the opening test will be `false`, and execution skips to the `else` outside the braces, producing the `Reply` `"My heart is in pieces on the floor"`. If `sentence` contains both `"love"` and `"you"`, the first test in the braces will be `true`, and the `Reply` will be `"I'm in heaven!"` The middle response `"But do you love me?"` will be triggered by a sentence that contains `"love"` but doesn't contain `"you"`, causing the first test in the braces to be `false`, and the `else` part in the braces to be executed.

Wrapper Classes

A *wrapper class* takes either an existing object or a value of primitive type, "wraps" or "boxes" it in an object, and provides a new set of methods for that type. The point of a wrapper class is to provide extended capabilities for the boxed quantity:

- It can be used in generic Java methods that require objects as parameters.
- It can be used in Java container classes like `ArrayList` that require the items be objects.

In each case, the wrapper class allows:

1. Construction of an object from a single value (wrapping or boxing the primitive in a wrapper object).
2. Retrieval of the primitive value (unwrapping or unboxing from the wrapper object).

Java provides a wrapper class for each of its primitive types. The two that you should know for the AP exam are the `Integer` and `Double` classes.

The `Integer` Class

The `Integer` class wraps a value of type `int` in an object. An object of type `Integer` contains just one instance variable whose type is `int`.

Here are the `Integer` methods and constants you should know for the AP exam. These are part of the Java Quick Reference.

```
Integer(int value)
```

Constructs an `Integer` object from an `int`. (Boxing)

```
int intValue()
```

Returns the value of this `Integer` as an `int`. (Unboxing)

```
Integer.MIN_VALUE
```

A constant equal to the minimum value represented by an `int` or `Integer`.

```
Integer.MAX_VALUE
```

A constant equal to the maximum value represented by an `int` or `Integer`.

The `Double` Class

The `Double` class wraps a value of type `double` in an object. An object of type `Double` contains just one instance variable whose type is `double`.

The methods you should know for the AP exam are analogous to those for type `Integer`. These, too, are part of the Java Quick Reference.

```
Double(double value)
```

Constructs a `Double` object from a `double`. (Boxing)

```
double doubleValue()
```

Returns the value of this `Double` as a `double`. (Unboxing)

Note

1. The `compareTo` and `equals` methods for the `Integer` and `Double` classes are no longer part of the AP Java subset. This is probably because the later versions of Java make extensive use of autoboxing and auto-unboxing.
2. `Integer` and `Double` objects are immutable. This means there are no mutator methods in the classes.

Autoboxing and Unboxing

This topic is part of the AP Java subset.

Autoboxing is the automatic conversion that the Java compiler makes between primitive types and their corresponding wrapper classes. This includes converting an `int` to an `Integer` and a `double` to a `Double`.

Autoboxing is applied when a primitive value is assigned to a variable of the corresponding wrapper class. For example,

```
Integer intOb = 3;   //3 is boxed

ArrayList<Integer> list = new ArrayList<Integer>();
list.add(4); //4 is boxed
```

Autoboxing also occurs when a primitive value is passed as a parameter to a method that expects an object of the corresponding wrapper class. For example,

```
public String stringMethod(Double d)
{ /* return string that has d embedded in it */ }

double realNum = 4.5;
String str = stringMethod(realNum);    //realNum is boxed
```

Unboxing is the automatic conversion that the Java compiler makes from the wrapper class to the primitive type. This includes converting an `Integer` to an `int` and a `Double` to a `double`.

Unboxing is applied when a wrapper class object is passed as a parameter to a method that expects a value of the corresponding primitive type. For example,

```
Integer intOb1 = 9;
Integer intOb2 = 8;

public static int sum (int num1, int num2)
{ return num1 + num2; }

System.out.println(sum(intOb1, intOb2)); //intOb1 and intOb2 are unboxed
```

Unboxing is also applied when a wrapper class object is assigned to a variable of the corresponding primitive type. For example,

```
int p = intOb1; //intOb1 is unboxed
```

Comparison of Wrapper Class Objects

Unboxing is often used in the comparison of wrapper objects of the same type. But it's trickier than it sounds. Don't use == to test `Integer` objects! You may get surprising results. The expression `if (intOb1 == intOb2)` tests whether `intOb1` and `intOb2` are the same *reference*. It does not test the actual values.

Example 1

```
Integer intOb1 = 4; //boxing
Integer intOb2 = 4; //boxing
if (intOb1 == intOb2)...
```

The test returns `true`, but not for the reason you might expect. The reason is that for efficiency Java creates only one Integer object if the `int` values are the same. So the references are the same. This is safe because an `Integer` cannot be altered. (It's immutable.)

Example 2

```
Integer intOb1 = 4;                //boxing
Integer intOb2 = new Integer(4);   //boxing
if (intOb1 == intOb2)...
```

This test returns `false` because use of `new` creates a new object. `intOb1` and `intOb2` are different references in this example. See the analogous situation for `String` objects in the Box on p. 176.

Example 3

```
Integer intOb1 = 4;  //boxing
int n = 4;
if (intOb1 == n)...
```

This test returns `true` because if the comparison is between an `Integer` object and a primitive integer type, the object is automatically unboxed.

Example 4

```
Integer intOb1 = 4;                //boxing
Integer intOb2 = new Integer(4);  //boxing
if (intOb1.intValue() == intOb2.intValue())...
```

This test returns `true` because the values of the objects are being tested. This is the correct way to test `Integer` objects for equality.

The relational operators less than (<) and greater than (>) do what you may expect when testing `Integer` and `Double` objects.

Example 5

```
Integer intOb1 = 4;       //boxing
Integer intOb2 = 8;       //boxing
if (intOb1 < intOb2)...
```

This test returns `true` because the compiler unboxes the objects as follows:

```
if (intOb1.intValue() < intOb2.intValue())
```

Example 6

```
Integer intOb1 = 4; //boxing
int n = 8;
if (intOb1 < n)...
```

This test will return `true` because, again, if one of the operands is a primitive type, the object will be unboxed.

The `Math` Class

This class implements standard mathematical functions such as absolute value, square root, trigonometric functions, the log function, the power function, and so on. It also contains mathematical constants such as π and e.

Here are the functions you should know for the AP exam:

```
static int abs(int x)
```

Returns the absolute value of integer x.

```
static double abs(double x)
```

Returns the absolute value of real number x.

```
static double pow(double base, double exp)
```

Returns $\texttt{base}^{\texttt{exp}}$. Assumes `base` > 0, or `base` = 0 and `exp` > 0, or `base` < 0 and `exp` is an integer.

```
static double sqrt(double x)
```

Returns $\sqrt{x}$, $x \geq 0$.

```
static double random()
```

Returns a random number r, where $0.0 \leq r < 1.0$. (See the next section, "Random Numbers.")

All of the functions and constants are implemented as static methods and variables, which means that there are no instances of `Math` objects. The methods are invoked using the class name, `Math`, followed by the dot operator.

Here are some examples of mathematical formulas and the equivalent Java statements.

1. The relationship between the radius and area of a circle is

$$r = \sqrt{A/\pi}$$

In code:

```
radius = Math.sqrt(area / Math.PI);
```

2. The amount of money A in an account after ten years, given an original deposit of P and an interest rate of 5% compounded annually, is

$$A = P(1.05)^{10}$$

In code:

```
a = p * Math.pow(1.05, 10);
```

3. The distance D between two points $P(x_P, y)$ and $Q(x_Q, y)$ on the same horizontal line is

$$D = |x_P - x_Q|$$

In code:

```
d = Math.abs(xp - xq);
```

Note

The static import construct allows you to use the static members of a class without the class name prefix. For example, the statement

```
import static java.lang.Math.*;
```

allows use of all `Math` methods and constants without the `Math` prefix. Thus, the statement in formula 1 above could be written

```
radius = sqrt(area / PI);
```

Static imports are not part of the AP subset.

Random Numbers

Random Reals

The statement

```
double r = Math.random();
```

produces a random real number in the range 0.0 to 1.0, where 0.0 is included and 1.0 is not.

This range can be scaled and shifted. On the AP exam you will be expected to write algebraic expressions involving `Math.random()` that represent linear transformations of the original interval $0.0 \le x < 1.0$.

Example 1

Produce a random real value x in the range $0.0 \le x < 6.0$.

```
double x = 6 * Math.random();
```

Example 2

Produce a random real value x in the range $2.0 \le x < 3.0$.

```
double x = Math.random() + 2;
```

Example 3

Produce a random real value x in the range $4.0 \le x < 6.0$.

```
double x = 2 * Math.random() + 4;
```

In general, to produce a random real value in the range `lowValue` $\le x <$ `highValue`:

```
double x = (highValue - lowValue) * Math.random() + lowValue;
```

Random Integers

Using a cast to `int`, a scaling factor, and a shifting value, `Math.random()` can be used to produce random integers in any range.

Example 1

Produce a random integer from 0 to 99.

```
int num = (int) (Math.random() * 100);
```

In general, the expression

```
(int) (Math.random() * k)
```

produces a random `int` in the range $0, 1, \ldots, k-1$, where k is called the scaling factor. Note that the cast to `int` truncates the real number `Math.random() * k`.

Example 2

Produce a random integer from 1 to 100.

```
int num = (int) (Math.random() * 100) + 1;
```

In general, if k is a scaling factor, and p is a shifting value, the statement

```
int n = (int) (Math.random() * k) + p;
```

produces a random integer n in the range $p, p+1, \ldots, p+(k-1)$.

Example 3

Produce a random integer from 5 to 24.

```
int num = (int) (Math.random() * 20) + 5;
```

Note that there are 20 possible integers from 5 to 24, inclusive.

Chapter Summary

All students should know about overriding the `equals` and `toString` methods of the `Object` class and should be familiar with the `Integer` and `Double` wrapper classes.

Know the AP subset methods of the `Math` class, especially the use of `Math.random()` for generating random numbers. Learn the `String` methods `substring` and `indexOf`, including knowing where exceptions are thrown in the `String` methods.