# Some Standard Classes

CHAPTER 4

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



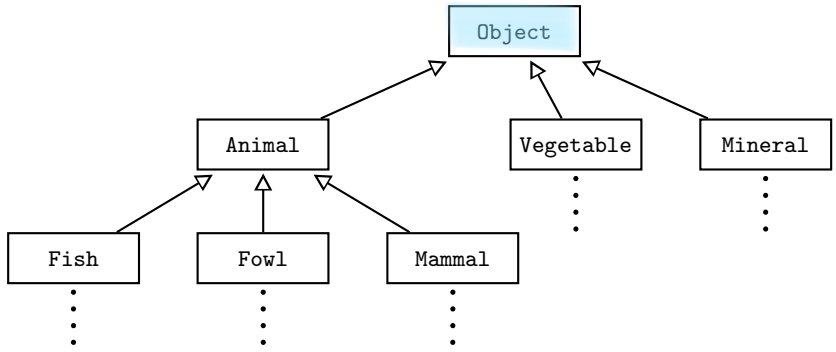
**Chapter Goals**

- The `Object` class
- The `String` class
- Wrapper classes
- The `Math` class
- Random numbers

## 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. Since `Object` is not an abstract class, all of its methods have implementations. 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 meaningless number (the address in memory of the object). 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 ...

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

ALWAYS THIS HEADER

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:

```
/** @return 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. 178).
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 overloaded in Java. To compare objects, one must use either the `equals` method or define a `compareTo` method for the class.

**Optional topic**

## THE `hashCode` METHOD

Every class inherits the `hashCode` method from `Object`. The value returned by `hashCode` is an integer produced by some formula that maps your object to an address in a hash table. A given object must always produce the same hash code. Also, two objects that are equal should produce the same hash code; that is, if `obj1.equals(obj2)` is true, then `obj1` and `obj2` should have the same hash code. Note that the opposite is not necessarily true. Hash codes do not have to be unique—two objects with the same hash code are not necessarily equal.

To maintain the condition that `obj1.equals(obj2)` is true implies that `obj1` and `obj2` have the same hash code, overriding `equals` means that you should override `hashCode` at the same time. You will not be required to do this on the AP exam.

You should, however, understand that every object is associated with an integer value called its hash code, and that objects that are equal have the same hash code.

*(continued)*

## 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 a mutated form of 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. 179).

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

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
```

## 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 that require the items be objects (see p. 242).

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 you should know for the AP exam:

`Integer(int value)`

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

`int compareTo(Integer other)`

Returns `0` if the value of this `Integer` is equal to the value of `other`, a negative integer if it is less than the value of `other`, and a positive integer if it is greater than the value of `other`.

`int intValue()`

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

`boolean equals(Object obj)`

Returns `true` if and only if this `Integer` has the same `int` value as `obj`.

NOTE

1. This method overrides `equals` in class `Object`.
2. This method throws a `ClassCastException` if `obj` is not an `Integer`.

`String toString()`

Returns a `String` representing the value of this `Integer`.

Here are some examples to illustrate the `Integer` methods:

```
Integer intObj = new Integer(6);  //boxes 6 in Integer object
int j = intObj.intValue();        //unboxes 6 from Integer object

System.out.println("Integer value is " + intObj);
//calls toString() for intObj
//output is
//Integer value is 6
```

```
Object object = new Integer(5);   //Integer is a subclass of Object

Integer intObj2 = new Integer(3);
int k = intObj2.intValue();
if (intObj.equals(intObj2))       //OK, evaluates to false
    ...
if (intObj.intValue() == intObj2.intValue())
    ...              //OK, since comparing primitive types

if (k.equals(j))     //error, k and j not objects
    ...
if ((intObj.intValue()).compareTo(intObj2.intValue()) < 0)
    ...              //error, can't use compareTo on primitive types

if (intObj.compareTo(object) < 0)  //Error. Parameter needs Integer cast
if (intObj.compareTo((Integer) object) < 0)  //OK
    ...
if (object.compareTo(intObj) < 0)  //error, no compareTo in Object
    ...
if (((Integer) object).compareTo(intObj) < 0)  //OK
    ...
```

## 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`.

`Double(double value)`

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

`double doubleValue()`

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

`int compareTo(Double other)`

Returns `0` if the value of this `Double` is equal to the value of `other`, a negative integer if it is less than the value of `other`, and a positive integer if it is greater than the value of `other`.

`boolean equals(Object obj)`

This method overrides `equals` in class `Object` and throws a `ClassCastException` if `obj` is not a `Double`. Otherwise it returns `true` if and only if this `Double` has the same `double` value as `obj`.

`String toString()`

Returns a `String` representing the value of this `Double`.

Here are some examples:

```
Double dObj = new Double(2.5);     //boxes 2.5 in Double object
double d = dObj.doubleValue();     //unboxes 2.5 from Double object

Object object = new Double(7.3);   //Double is a subclass of Object
Object intObj = new Integer(4);
if (dObj.compareTo(object) > 0) //Error. Parameter needs cast to Double
if (dObj.compareTo((Double) object) > 0)    //OK
    ...
if (dObj.compareTo(intObj) > 0)    //ClassCastException
    ...                            //can't compare Integer to Double
```

Remember: `Integer`, `Double`, and `String` all have a `compareTo` method.

NOTE

1. `Integer` and `Double` objects are immutable: There are no mutator methods in the classes.
2. See p. 242 for a discussion of auto-boxing and -unboxing. This useful feature will *not* be tested on the AP exam.

## 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 $\pi$ 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:

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

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 $0.0 \leq x < 6.0$.

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

**Example 2**

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

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

**Example 3**

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

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

In general, to produce a random real value in the range `lowValue` $\leq x <$ `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.

## NOTE

There is further discussion of strings and random numbers, plus additional questions, in Chapter 9 (The AP Computer Science Labs).

## 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 integers. Know the `String` methods `substring` and `indexOf` like the back of your hand, including knowing where exceptions are thrown in the `String` methods.