A method can call other methods of the same object or of a different object. For example, `Balloon`'s `isOnBorder` method calls the same balloon's `distance` method:

```
public boolean isOnBorder(int x, int y)
{
  double d = distance(x, y);
  return d >= 0.9 * radius && d <= 1.1 * radius;
}
```

`Balloon`'s `draw` method calls `Graphics`'s methods `fillOval` or `drawOval`:

```
public void draw(Graphics g, boolean makeItFilled)
{
  g.setColor(color);
  if (makeItFilled)
    g.fillOval(xCenter - radius,
               yCenter - radius, 2*radius, 2*radius);
  else
    g.drawOval(xCenter - radius,
               yCenter - radius, 2*radius, 2*radius);
}
```

❖ ❖ ❖

The `Balloon` class provides a well-defined functionality through its constructors and public methods. The user of the `Balloon` class (possibly a different programmer) does not need to know all the details of how the class `Balloon` works, only how to construct its objects and what they can do. In fact, all the instance variables in `Balloon` are declared `private` so that programmers writing classes that use `Balloon` cannot refer to them directly. Some of class's methods can be declared `private`, too. This technique is called *encapsulation* and *information hiding*.

There are two advantages to such an arrangement:

- `Balloon`'s programmer can change the structure of the fields in the `Balloon` class, and the rest of the project won't be affected, as long as `Balloon`'s constructors and public methods have the same specifications and work as before;
- `Balloon`'s programmer can document the `Balloon` class for the other team members (and other programmers who want to use this class) by describing all its constructors and public methods; there is no need to document the implementation details.

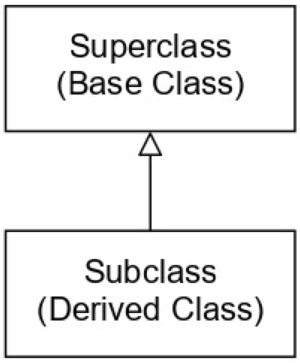
It is easier to maintain, document, and reuse an encapsulated class.

After a class is written, it is a good idea to test it in isolation from other classes. We can create a small program that tests some of `Balloon`'s features. For example:

```
import java.awt.Color;

public class TestBalloon
{
  public static void main(String[] args)
  {
    // Create a Balloon called greenie, centered at x = 50, y = 100
    // with radius 30 and color Color.GREEN:

    ___________< statement not shown >___________ ;

    System.out.println("x = " + greenie.getX());
    System.out.println("y = " + greenie.getY());
    System.out.println("radius = " + greenie.getRadius());

    // Call greenie's move method to move its center to (60, 110):

    ___________< statement not shown >___________ ;

    System.out.println("x = " + greenie.getX());
    System.out.println("y = " + greenie.getY());

    // Call greenie's isOnBorder method to see if (81, 131)
    // is on its border:

    boolean onBorder = ________< expression not shown >________ ;

    System.out.println("(81, 131) is on the border: " + onBorder);
  }
}
```

Type in the above code, filling in the blanks, and save it in the `TestBalloon.java` file. Create a project that includes the `TestBalloon.java` and `Balloon.java` files and test your program. Explain the output. Now try to call `move` with the parameters 50, 100. Explain the output.

If class *D* extends class *B*, then *B* is called a *superclass* (or a *base class*) and *D* is called a *subclass* (or a *derived class*). The relationship of inheritance is usually indicated in UML diagrams by an arrow with a triangular head from a subclass to its superclass:

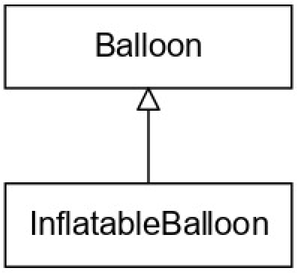


In Java you can extend a class without having its source code.

A subclass *inherits* all the methods and fields of its superclass. Constructors are not inherited; a subclass can provide its own if it needs a constructor.

In Java every class extends the library class `Object` by default. `Object` supplies a few common methods, including `toString` and `getClass`.

In our example, we create a new class `InflatableBalloon`, which extends `Balloon`:



`InflatableBalloon` can be a very short class:

```
import java.awt.Color;

/**
 * Represents an inflatable balloon
 */
public class InflatableBalloon extends Balloon
{
       < Constructors may be needed, not shown >

  public void inflate(int percentage)
  {
       < code not shown >
  }
}
```

We can provide constructors to this class or we can just let the default no-args constructor do the job. It will call automatically `Balloon`'s no-args constructor. `InflatableBalloon` adds one method, `inflate`.

❖ ❖ ❖

Notice the following paradox. Our `InflatableBalloon` class inherits all the fields from `Balloon`. So an `InflatableBalloon` has a `radius` field. However, this and other fields are declared `private` in `Balloon`. This means that the programmer who wrote the `InflatableBalloon` class did not have direct access to them (even if he is the same programmer who wrote `Balloon`!). Recall that the `Balloon` class is fully encapsulated and all its fields are private. So the statement

```
     radius = (int)Math.round(radius * (1 + 0.01*percentage));
```

won't work in `InflatableBalloon`. Try it and see for yourself. What do we do? `Balloon`'s subclasses and, in fact, any other classes that use `Balloon` might need access to the values stored in its private fields.

To resolve the issue, the `Balloon` class provides public methods that simply return the values of its private fields: `getX`, `getY`, `getRadius`, `getColor`.

Such methods are called *accessor* methods (or simply *accessors*) or *getters* because they give outsiders access to the values of private fields of an object.

It is a very common practice to provide accessor methods for those private fields of a class that may be of interest to other classes. (You will have to figure out how this works for our `inflate` method in Chapter 5 exercises, Question 17.) Methods like `setRadius` are called *setters* or *modifiers*.

❖ ❖ ❖

Inheritance represents the *IS-A relationship* between objects. If RoundBalloon is a subclass of Balloon, then a RoundBalloon IS-A (is a) Balloon. In addition to the fields and methods, an object of a subclass inherits a less tangible but also very valuable asset from its superclass: its type. It is like inheriting the family name or title. The superclass's type becomes a secondary, more generic type of an object of the subclass. Whenever a statement or a method call expects a Balloon-type object, you can plug in a RoundBalloon-type object instead.

For example, the class DrawingPanel has a statement:

```
      balloons.add(activeBalloon);
```

It expects that activeBalloon refers to a Balloon object, but it can refer to a RoundBalloon object as well. The following statement will compile with no problems:

```
    Balloon round = new RoundBalloon();
```

Since every class extends the class Object by default, every object IS-A(n) Object.

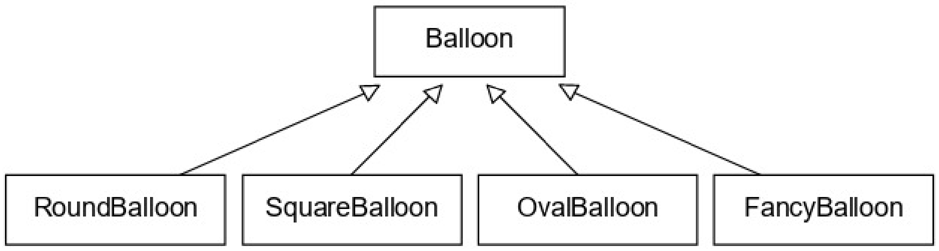
Inheritance allows us to add objects of various subclasses to a project. For example, in the *BalloonDraw* project we can define subclasses of Balloon: RoundBalloon, OvalBalloon, SquareBalloon, or FancyBalloon (you will do just that in the lab in the next section). These subclasses will take advantage of some of the methods of Balloon, but some other methods, such as draw, will have to be redefined appropriately for each subclass.

❖ ❖ ❖

We have made our first steps in OOP, but a lot remains to be learned. We will discuss more advanced concepts in Chapters 10 and 12.

4.6 *Case Study and Lab:* Balloons of All Kinds

The purpose of this lab is to add support for balloons of different shapes to the *BalloonDraw* program. Different balloons will be implemented as subclasses of the `Balloon` class:



We have updated the `ControlPanel` class, replacing the "Add Balloon" button with a pull-down list, which allows the user to pick a balloon of a particular shape. We have also added a parameter to the `addBalloon` method in the `DrawingPanel` class. It indicates what kind of balloon to add to the list of balloons. Your task is to write classes for different kinds of balloons.

1. Make a copy of `JM\Ch04\BalloonDraw\Balloon.java` and name it `RoundBalloon.java`. Replace the class header in `RoundBalloon` to read

```
public class RoundBalloon extends Balloon
```

   Remove all the fields. Rename the constructors appropriately and replace the code in them with calls to `super` as follows:

```
public RoundBalloon()
{
  super();  // this is optional: default
}

public RoundBalloon(int x, int y, int r, Color c)
{
  super(x, y, r, c);
}
```

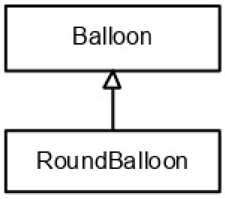
   `super` calls the corresponding constructor of the superclass.

Remove all the methods from `RoundBalloon.java` except `draw`. Go back to the `Balloon` class and remove the `draw` method, leaving only empty braces:

```
public void draw(Graphics g, boolean makeItFilled)
{
}
```

There is a better way to "disable" a method: we can declare it "abstract" and provide no code for it at all, not even empty braces. We will discuss abstract classes and methods in Chapter 12.

We now have a class `Balloon` and its subclass `RoundBalloon`.



`Balloon`'s private fields are not directly accessible in `RoundBalloon`, so you need to replace the references to them in the `draw` method with calls to "getter" methods, as explained on Page 83. It may be convenient to introduce temporary "local" variables to hold the values obtained with getters:

```
  ...
    int x = getX();
    int y = getY();
    int r = getRadius();
    ...
      g.fillOval(x - r, y - r, 2*r, 2*r);
```

Notice how `Balloon` and its subclass share responsibilities: `Balloon` supplies more general methods, while `RoundBalloon` supplies a `draw` method specific to the round shape.

At this point it would be nice to test what we have got, but there is a problem: the program also expects `OvalBalloon`, `SquareBalloon`, and `FancyBalloon` classes and we don't have them. The solution is to use temporary "stub" classes. For example:

```
import java.awt.Color;

public class OvalBalloon extends Balloon
{
  public OvalBalloon() {  }

  public OvalBalloon(int x, int y, int r, Color c)
  { super(x, y, r, c); }
}
```

Type up the above stub class for `OvalBalloon` and create similar stub classes for `SquareBalloon` and `FancyBalloon`.

Set up a project with the following eight classes in it: `Balloon.java`, `RoundBalloon.java`, your three stub classes, and the three classes provided for you in the `JM\Ch04\Balloons-All-Kinds` folder — `BalloonDraw.java`, `ControlPanel.java`, and `DrawingPanel.java`.

Test your program; make sure it runs the same way as the original *BalloonDraw* when you choose "Round" from the pull-down list. What happens when you choose "Oval" or another balloon that has not been fully implemented yet? Explain why.

2. Replace the `OvalBalloon` stub class with a real class. The easiest way of doing this is to adapt the code from the `RoundBalloon` class. Copy `RoundBalloon.java` into `OvalBalloon.java`, rename the class and the constructors, and change the `draw` method. An oval balloon is twice as tall as it is wide; its height is `2*radius`, same as before, but its width is `radius`. When you draw it, make sure its center remains at `xCenter`, `yCenter`.

   You also need to redefine the `distance` method. Copy it from the `Balloon` class into `OvalBalloon` and make a simple change, so that the "distance" from any point on the border is equal to `radius`. Hint: if you stretch the oval horizontally by a factor of two, it becomes a circle, so when you define the "distance," you need to multiply `dx` by 2.

   Retest your program thoroughly. Make sure you can "grab" an oval balloon at any point inside it to move it, and "grab" the balloon at any point on the border to stretch it.

3.■ Repeat Step 2 for the `SquareBalloon` class. Explore the documentation for the Java library class `Graphics` and find the methods that draw and fill a rectangle. The parameters for drawing a square will be the same as the parameters for drawing a circle (because the library methods that draw an oval use the oval's bounding rectangle).

   You need to redefine the `distance` method again. It turns out there is a simple formula that will make the "distance" from any point on the border of the square equal to radius. Hint: `Math.abs(dx)` returns the absolute value of `dx`; `Math.max(dx, dy)` returns the larger of `dx` and `dy`. Retest your program thoroughly again.

Continued ⇨

4.◆ Create a `FancyBalloon` class that has a shape of your own design. One example may be a vertically stretched rectangle with rounded corners (see the `Graphics` class). Another example is a diamond shape. For that you will need to use the `drawPolygon` method that takes a list of vertices. (There is a simple "distance" formula for a diamond.) Another way to create an interesting shape is to take two overlapping shapes with the same center, for example, two ovals, one stretched vertically, the other horizontally, or a square and a diamond. If you know the "distance" formula for each shape, you can combine them to make the "distance" for their combination. Hint: `Math.min(d1, d2)` returns the smaller of `d1` and `d2`.

You can also play with colors, mixing different colors and adding decorations to your balloon. Refer to the Java documentation for the `Color` class. If `c` is a `Color`, `c.darker()` returns a darker color of the same tint; `c.brighter()` returns a lighter color.

4.7 Summary

An OOP program is best visualized as a virtual world of interacting objects. A program's source code describes different types of objects used in the program. Objects of the same type are said to belong to the same *class*. An object is called an *instance* of its class. The source code of a Java program consists of *definitions of classes*.

The source code for each class is stored in a separate file with the same name as the class and the extension `.java`. By convention, a class name always starts with a capital letter. It is customary to place all your classes for a small project into the same folder. Several compiled Java classes may be collected in one `.jar` file.

A *CRC card* gives a preliminary, informal description of a class, listing its name, the key "responsibilities" of its objects, and the other classes this class depends on ("collaborators").

The `import` statements at the top of a class's source code tell the compiler where it can find the library classes and packages used in that class.

A class's source code begins with an optional brief comment that describes the purpose of the class, followed by `import` statements, if necessary, then the class's header, and the class's body within braces. A class defines the data elements of an object of that class, called *instance variables* or *fields*. Each instance variable has a name, given by a programmer, and a type. The set of fields serves as the "personal memory" of an object. Their values may be different for different objects of the class, and these values can change while the program is running. A class also defines *constructors*, which are short procedures for creating objects of that class, and *methods*, which describe what an object can do.

A constructor always has the same name as its class. A constructor is used primarily to set the initial values of the object's fields. It can accept one or several parameters that are used to initialize the fields. A constructor that does not take any parameters is called a *no-args constructor*. A class may have several constructors that differ in the number or types of parameters that they accept. If no constructors are defined, the compiler automatically supplies one default no-args constructor that sets all the instance variables to default values (zeroes for numbers, `null` for objects, `false` for `boolean` variables).

You create a new object in the program using the `new` operator. The parameters passed to `new` must match the number, types, and order of parameters of one of the constructors in the object's class, and `new` invokes that constructor. `new` allocates memory to store the newly constructed object.

The functionality of a class — what its objects can do — is defined by its *methods*. A method accomplishes a certain task. It can be called from constructors and other methods of the same class and, if it is declared `public`, from constructors and methods of other classes. A method can take parameters as its "inputs." Parameters passed to a method must match the number, types, and order of parameters that the method expects. A method can return a value of a specified type to the caller. A method declared `void` does not return any value.

In OOP, all the instance variables of a class are usually declared `private`, so only objects of the same class have direct access to them. Some of the methods may be private, too. Users of a class do not need to know how the class is implemented and what its private fields and methods are. This practice is called *information hiding*. A class interacts with other classes only through a well-defined set of constructors and public methods. This concept is called *encapsulation*. Encapsulation facilitates program maintenance, code reuse, and documentation. A class often provides public methods that return the values of an object's private fields, so that an object of a different class can access those values. Such methods are called *accessor methods*, *accessors*, or "getters." Methods that set the values of private fields are called *modifiers* or "setters."

A class definition does not have to start from scratch: it can *extend* the definition of another class, adding fields and/or methods and/or overriding (redefining) some of the methods. This concept is called *inheritance*. It is said that a *subclass* (or *derived class*) extends a *superclass* (or *base class*). Constructors are not inherited.

An object of a subclass also inherits the type of its superclass as a secondary, more generic type. This formalizes the *IS-A relationship* between objects: an object of a subclass IS-A(n) object of its superclass.

Exercises

Sections 4.1-4.4

1. Mark true or false and explain:

(a) The name of a class in Java must be the same as the name of its source file (excluding the extension `.java`). _____

(b) The names of classes are case-sensitive. _____

(c) The `import` statement tells the compiler which other classes use this class. _____ ✓

2. Mark true or false and explain:

(a) The *BalloonDraw* program consists of one class. ______ ✓

(b) A Java program can have as many classes as necessary. ______

(c) A Java program is allowed to create only one object of each class. ______

(d) Every class has a method called `main`. ______ ✓

3. Navigate your browser to Oracle's Java API (Application Programming Interface) documentation web site (for example, `http://download.oracle.com/javase/`, or simply google "Java API"). If you have the Java documentation installed on your computer, open the file `<JDK folder>/docs/api/index.html` (for example, `C:/Program Files/Java/jdk-16/docs/api/index.html`)

(a) Approximately how many different packages are listed in the API spec?

(b) Find `JFrame` in the list of classes in the left column and click on it. Scroll down the main window to the "Method Summary" section. Approximately how many methods does the `JFrame` class have, including methods inherited from other classes? 3? 12? 25? 300-400? ✓

4. Mark true or false and explain:

(a) Fields of a class are usually declared `private`. ______
(b) An object has to be created before it can be used. ______ ✓
(c) A class may have more than one constructor. ______
(d) The programmer gives names to objects in his program. ______
(e) When an object is created, the program always calls its `init` method. ______ ✓

5. What are the benefits of encapsulation? Name three.

6. Make a copy of `balloondraw.jar` (`JM\Ch04\BalloonDraw\balloondraw.jar`) and rename it into `balloondraw.zip`. Examine its contents. As you can see, a `.jar` file is nothing more than a compressed folder. How many compiled Java classes does it hold?

7.◆ Create a class `Book` with two private `int` fields, `numPages` and `currentPage`. Supply a constructor that takes one parameter and sets `numPages` to that value and `currentPage` to 1. Provide accessor methods for both fields. Also provide a method `nextPage` that increments `currentPage` by 1, but only if `currentPage` is less than `numPages`.

⋵ Hint:

```
if (currentPage < numPages)
    currentPage++;
```
⋾

Create a `BookTest` class with a `main` method. Let `main` create a `Book` object with 3 pages, then call its `nextPage` method three times, printing out the value of `currentPage` after each call.

8.■ (a) Write a simple class `Circle` with one field, `radius`. Supply one constructor that takes the radius of the circle as a parameter:

```
public Circle(int r)
{
  ...
}
```

Supply one method, `getArea`, that computes and returns (not prints!) the area of the circle. Provide the accessor method `getRadius`. Create a small class `CircleTest` with a `main` method that prompts the user to enter an integer value for the radius, creates one `Circle` object with that radius, calls its `getArea` method, and prints out the returned value.

⋞ Hints:

1. `getArea` should return a value of the type `double`;
2. The class `Math` has a constant `Math.PI`;
3. Figure out a way to square a number: search online or look in the `Math` class, or just do it.

⋟

(b) Create a class `Cylinder` with two private fields: `Circle base` and `int height`. Is it fair to say that a `Cylinder` HAS-A `Circle`? Provide a constructor that takes two `int` parameters, `r` and `h`, initializes `base` to a new `Circle` with radius `r`, and initializes `height` to `h`. Provide a method `getVolume` that returns the volume of the cylinder (which is equal to the base area multiplied by height). Create a simple test program `CylinderTest`, that prompts the user to enter the radius and height of a cylinder, creates a new cylinder with these dimensions, and displays its volume.

9.◆ JM\Ch04\Exercises\CoinTest.java is part of a program that shows a picture of a coin in the middle of a window and "flips" the coin every two seconds. Your task is to supply the second class for this program, Coin.

The Coin class should have one constructor that takes two parameters of the type Image: the heads and tails pictures of the coin. The constructor saves these images in the coin's private fields (of the type Image). Coin should also have a field that indicates which side of the coin is displayed. The Coin class should have two methods:

```
/**
 * Flips this coin
 */
public void flip()
{
  ...
}
```

and

```
/**
 * Draws the appropriate side of the coin
 * centered at (x, y)
 */
public void draw(Graphics g, int x, int y)
{
  ...
}
```

⋞ Hints:

1.
```
import java.awt.Image;
import java.awt.Graphics;
```

2. The class Graphics has a method that draws an image at a given location. Call it like this:

```
        g.drawImage(pic, xUL, yUL, null);
```

   where pic is the image and xUL and yUL are the coordinates of its upper left corner. You need to calculate xUL and yUL from the x and y passed to Coin's draw. Explore the documentation for the library class Image to find methods that return the width and height of an image.

3. Find copyright-free image files for the two sides of a coin on the Internet or scan or take a picture of a coin and create your own image files.

⋟

Sections 4.5-4.7

10. Mark true or false:

(a) A subclass inherits all the fields and public methods of its superclass. _____ ✓

(b) A subclass inherits all those constructors of its superclass that are not defined explicitly in the subclass. _____ ✓

11. Question 8-b above asks you to write a class `Cylinder` with two fields: `Circle base` and `double height`. Instead of making `base` a field we could simply derive `Cylinder` from `Circle`, adding only the `height` field. Discuss the merits of this design option. ✓

12. The classes in this question refer to the Lab 4.6. Which of the following assignment statements will compile without errors? Explain your answers.

(a) `RoundBalloon b1 = new RoundBalloon();` ______

(b) `Balloon b2 = new RoundBalloon(100, 100, 10, Color.RED);` ______

(c) `Balloon b3  = new SquareBalloon();` ______

(d)■ `RoundBalloon b5 = new Balloon();` ______

13.■ Using the `Balloon` class in `JM\Ch04\BalloonDraw\Balloon.java`, write a simple console application that creates a `Balloon` and "prints it out":

```
Balloon b = new Balloon(100, 100, 20, Color.RED);
System.out.println(b);
```

What is displayed? It appears the program doesn't know how to properly "print" a `Balloon` object. Find online how to fix that by overriding `Object`'s `toString` method in the `Balloon` class. ⟨ Hint: see, for example, `www.javatpoint.com/understanding-toString()-method`. ⟩ ✓

14.◆ Using the `Balloon` class and its subclasses defined in the lab in Section 4.6, add a couple of balloons of different shapes and colors to the *Banner* program in Chapter 2 (`JM\Ch02\HelloGui\Banner.java`). Make them fly up, then start at the bottom again. ⟨ Hints: add fields that refer to the balloons; create the balloons in the `main` method; draw the balloons in the `paintComponent` method; a `Balloon` object "knows" its own position; increase the height of the window. ⟩

`int chapter = 5;`

Data Types, Variables, and Arithmetic

5.1 Prologue

Java and other high-level languages let programmers refer to a memory location by name. These named "containers" for values are called *variables*. The programmer gives variables meaningful names that reflect their role in the program. The compiler/interpreter takes care of all the details — allocating memory space for the variables and representing data in the computer memory.

The term "variable" is borrowed from algebra because, as in algebra, variables can assume different values and can be used in *expressions*. The analogy ends there, however. In a computer program, variables are actively manipulated by the program. A variable is like a slate on which the program can write a new value when necessary and from which it can read the current value. For example, the statement

```
    a = b + c;
```

does not mean that a is equal to $b + c$, but rather a set of instructions:

1. Get the current value of `b`;
2. Get the current value of `c`;
3. Add the two values;
4. Assign the result to `a` (write the result into `a`).

The same is true for

```
    a = 4 - a;
```

It is not an equation, but a set of instructions for changing the value of `a`:

1. Get the current value of the variable `a`;
2. Subtract it from `4`;
3. Assign the result back to `a` (write the new value into `a`).

In Java, a statement

```
    someName = expression;
```

represents an *assignment* operation that evaluates (finds the value of) the expression on the right side of the = sign and assigns that value to (writes it into) the variable `someName` on the left side of =. The = sign is read "gets the value of": "`someName` gets the value of *expression*." (If you want to compare two values, use another operator, ==, to mean "is equal to.")

In Java, every variable has a *data type*. This can be either a *primitive data type* (such as `int`, `double`, `boolean`, and `char`) or a class type (such as `String`, `Color`, `Balloon`). The programmer specifies the variable's data type based on the kind of data it will contain.

A variable's data type determines the range of values that can be stored in the variable, the amount of space allocated for it in memory, and the kind of operations that can be performed with it.

A variable of type `int`, for example, contains an integer value, which can be in the range from -2^{31} to $2^{31}-1$; a variable of type `double` represents a real number. A variable of type `String` refers to an object of the `String` class.

In Java, a variable that represents an object holds a *reference* to that object. A reference is basically the address of the object in RAM when the program is running. When an object is created with the `new` operator, the interpreter allocates memory space for the object and returns a reference to it. That reference can be saved in a variable and used to access the object. We'll explain references in more detail in Section 10.4.

In this chapter we explain the following concepts and elements of Java syntax:

- The syntax and placement of declarations for variables and constants
- Primitive data types
- Strings
- Literal and symbolic constants
- Conversion of values from one type to another (casts)
- The scope of variables and symbolic constants
- Java's arithmetic operators