And so even though we face the difficulties of today and tomorrow, I still have a dream. It is a dream deeply rooted in the American dream. I have a dream that one day this nation will rise up and live out the true meaning of its creed: "We hold these truths to be self-evident, that all men are created equal."

Martin Luther King, Jr.
“I Have a Dream”
Gates B-12
Monday, January 18
3:15 P.M.

The GObject Hierarchy
The classes that represent graphical objects form a hierarchy, part of which looks like this:

```
GObject
  ↓
GLabel, GRect, GOval, GLine
```

The GObject class represents the collection of all graphical objects. The four subclasses shown in this diagram correspond to particular types of objects: labels, rectangles, ovals, and lines. The class diagram makes it clear that any GLabel, GRect, GOval, or GLine is also a GObject.

Operations on the GObject Class
The following operations apply to all GObject:

```
object.setColor(color)
Sets the color of the object to the specified color constant.
```

```
object.setLocation(x, y)
Changes the location of the object to the point (x, y).
```

```
object.move(dx, dy)
Moves the object on the screen by adding dx and dy to its current coordinates.
```

The standard color names are defined in the java.awt package:

```
Color.BLACK
Color.DARK_GRAY
Color.GRAY
Color.LIGHT_GRAY
```

```
Color.WHITE
Color.RED
Color.YELLOW
Color.GREEN
Color.CYAN
```

```
Color.BLUE
Color.MAGENTA
Color.ORANGE
Color.PINK
```

Operations on the GLabel Class

Constructor
```
new GLabel(text, x, y)
```
Creates a label containing the specified text that begins at the point (x, y).

Methods specific to the GLabel class

```
label.setFont(font)
Sets the font used to display the label as specified by the font string.
```

The font is typically specified as a string in the form

```
*family-style-size*
```

where

```
family
```

is the name of a font family,

```
styple
```

is either PLAIN, BOLD, ITALIC, or BOLDITALIC,

```
size
```

is an integer indicating the point size.

Drawing Geometrical Objects

Constructors

```
new GRect(x, y, width, height)
```
Creates a rectangle whose upper left corner is at (x, y) of the specified size.

```
new GOval(x, y, width, height)
```
Creates an oval that fits inside the rectangle with the same dimensions.

```
new GLine(x0, y0, x1, y1)
```
Creates a line extending from (x0, y0) to (x1, y1).

Methods shared by the GRect and GOval classes

```
object.setFilled(fill)
```
If fill is true, fills in the interior of the object; if false, shows only the outline.

```
object.setFillColor(color)
```
Sets the color used to fill the interior, which can be different from the border.
Statement Types in Java

- Programs in Java consist of a set of classes. Those classes contain methods, and each of those methods consists of a sequence of statements.
- Statements in Java fall into three basic types:
  - Simple statements
  - Compound statements
  - Control statements

Simple statements are formed by adding a semicolon to the end of a Java expression.

Compound statements (also called blocks) are sequences of statements enclosed in curly braces.

Control statements fall into two categories:
- Conditional statements that specify some kind of test
- Iterative statements that specify repetition

Boolean Expressions

In many ways, the most important primitive type in Java is boolean, even though it is by far the simplest. The only values in the boolean domain are true and false, but these are exactly the values you need if you want your program to make decisions.

The name boolean comes from the English mathematician George Boole who in 1854 wrote a book entitled *An Investigation into the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities*. That book introduced a system of logic that has come to be known as Boolean algebra, which is the foundation for the boolean data type.

Boolean Operators

- The operators used with the boolean data type fall into two categories: relational operators and logical operators.
- There are six relational operators that compare values of other types and produce a boolean result:
  - `==` Equals
  - `!=` Not equals
  - `<` Less than
  - `>` Greater than
  - `<=` Less than or equal to
  - `>=` Greater than or equal to

For example, the expression `n <= 10` has the value true if `x` is less than or equal to 10 and the value false otherwise.

There are also three logical operators:
- `&&` Logical AND
- `||` Logical OR
- `!` Logical NOT

Short-Circuit Evaluation

- Java evaluates the `&&` and `||` operators using a strategy called short-circuit mode in which it evaluates the right operand only if it needs to do so.
- For example, if `n` is 0, the right hand operand of `&&` in `n != 0 && x / n` is not evaluated at all because `n != 0` is false. Because the expression `false && anything` is always false, the rest of the expression no longer matters.
- One of the advantages of short-circuit evaluation is that you can use `&&` and `||` to prevent execution errors. If `n` were 0 in the earlier example, evaluating `x / n` would cause a "division by zero" error.

Notes on the Boolean Operators

- Remember that Java uses `=` to denote assignment. To test whether two values are equal, you must use the `==` operator.
- It is not legal in Java to use more than one relational operator in a single comparison as is often done in mathematics. To express the idea embodied in the mathematical expression `0 <= x && x <= 9` you need to make both comparisons explicit, as in `0 <= x && x <= 9`

- The `||` operator means either or both, which is not always clear in the English interpretation of or.
- Be careful when you combine the `!` operator with `&&` and `||` because the interpretation often differs from informal English.

The if Statement

The simplest of the control statements is the `if` statement, which occurs in two forms. You use the first form whenever you need to perform an operation only if a particular condition is true:

```java
if (condition) { statements to be executed if the condition is true }
```

You use the second form whenever you want to choose between two alternative paths, one for cases in which a condition is true and a second for cases in which that condition is false:

```java
if (condition) { statements to be executed if the condition is true }
else { statements to be executed if the condition is false }
```
Common Forms of the if Statement

The examples in the book use only the following forms of the if statement:

- Single line if statement
  ```java
  if (condition) statement
  ```

- Multiline if statement with curly braces
  ```java
  if (condition) {
  statement
  . . . more statements . . .
  }
  ```

- If/else statement with curly braces
  ```java
  if (condition) {
  statements
  true
  } else {
  statements
  false
  }
  ```

- Cascading if statement
  ```java
  if (condition)
  statements
  else if (condition)
  statements
  . . . more if/else conditions . . .
  } else {
  statements
  else
  }
  ```

The examples in the book use only the following forms of the if statement:

The ?:: Operator

- In addition to the if statement, Java provides a more compact way to express conditional execution that can be extremely useful in certain situations. This feature is called the ?:: operator (pronounced question-mark-colon) and is part of the expression structure. The ?:: operator has the following form:

  ```java
  condition ? expression1 : expression2
  ```

  - When Java evaluates the ?:: operator, it first determines the value of `condition`, which must be a boolean. If `condition` is true, Java evaluates `expression1` and uses that as the value; if `condition` is false, Java evaluates `expression2` instead.

  ```java
  condition ? expression1 : expression2
  ```

  - You could use the ?:: operator to assign the larger of `x` and `y` to the variable `max` like this:

    ```java
    max = (x > y) ? x : y;
    ```

The switch Statement

- The switch statement provides a convenient syntax for choosing among a set of possible paths:

  ```java
  switch (expression) {
  case v1:
  statements to be executed if expression = v1
  break;
  case v2:
  statements to be executed if expression = v2
  break;
  . . . more case clauses if needed . . .
  default:
  statements to be executed if no values match
  break;
  }
  ```

  - The switch statement is useful when the program must choose among several cases, as in the following example:

    ```java
    public void run() {
    println("This program shows the number of days in a month.");
    int month = readInt("Enter numeric month (Jan=1): ");
    switch (month) {
    case 2:
    println("28 days (29 in leap years)"� break;
    case 4: case 6: case 9: case 11:
    println("30 days") break;
    case 1: case 3: case 5: case 7: case 8: case 12:
    println("31 days") break;
    default:
    println("Illegal month number") break;
    }
    }
    ```

The while Statement

- The while statement is the simplest of Java’s iterative control statements and has the following form:

  ```java
  while (condition) {
  statements to be repeated
  }
  ```

  When Java encounters a while statement, it begins by evaluating the condition in parentheses, which must have a boolean value. If the value of condition is true, Java executes the statements in the body of the loop.

  At the end of each cycle, Java reevaluates condition to see whether its value has changed. If condition evaluates to false, Java exits from the loop and continues with the statement following the closing brace at the end of the while body.

The DigitSum Program

- This program sums the digits in an integer:

  ```java
  public void run() {
  println("This program sums the digits in an integer.");
  int n = readInt("Enter a positive integer:");
  int dsum = 0;
  while (n > 0) {
  dsum += n % 10;
  n /= 10;
  }
  println("The sum of the digits is " + dsum);
  }
  ```

Example of the switch Statement

- The switch statement is useful when the program must choose among several cases, as in the following example:

  ```java
  public void run() {
  println("This program shows the number of days in a month.");
  int month = readInt("Enter numeric month (Jan=1): ");
  switch (month) {
  case 2:
  println("28 days (29 in leap years)"� break;
  case 4: case 6: case 9: case 11:
  println("30 days") break;
  case 1: case 3: case 5: case 7: case 8: case 12:
  println("31 days") break;
  default:
  println("Illegal month number") break;
  }
  }
  ```
The for Statement

The for statement in Java is a particularly powerful tool for specifying the control structure of a loop independently from the operations the loop body performs. The syntax looks like this:

```
for (init; test; step) {
    statements to be repeated
}
```

Java evaluates a for statement by executing the following steps:
1. Evaluate `init`, which typically declares a control variable.
2. Evaluate `test` and exit from the loop if the value is `false`.
3. Execute the statements in the body of the loop.
4. Evaluate `step`, which usually updates the control variable.
5. Return to step 2 to begin the next loop cycle.

Comparing for and while

The for statement

```
for (init; test; step) {
    statements to be repeated
}
```

is functionally equivalent to the following code using while:

```
init;
while (test) {
    statements to be repeated
    step;
}
```

The advantage of the for statement is that everything you need to know to understand how many times the loop will run is explicitly included in the header line.

Exercise: Reading for Statements

Describe the effect of each of the following for statements:
1. `for (int i = 1; i <= 10; i++)`
2. `for (int i = 0; i < N; i++)`
3. `for (int n = 99; n >= 1; n -= 2)`
4. `for (int x = 1; x <= 1024; x *= 2)`

The Checkerboard Program

```
public void run() {
    double sqSize = (double) getHeight() / N_ROWS;
    for (int i = 0; i < N_ROWS; i++) {
        for (int j = 0; j < N_COLUMNS; j++) {
            double x = j * sqSize;
            double y = i * sqSize;
            GRect sq = new GRect(x, y, sqSize, sqSize);
            sq.setFilled((i + j) % 2 != 0);
            add(sq);
        }
    }
}
```

Simple Graphical Animation

The while and for statements make it possible to implement simple graphical animation. The basic strategy is to create a set of graphical objects and then execute the following loop:

```
for (int i = 0; i < N_STEPS; i++) {
    update the graphical objects by a small amount
    pause(PAUSE_TIME);
}
```

On each cycle of the loop, this pattern updates each animated object by moving it slightly or changing some other property of the object, such as its color. Each cycle is called a time step.

After each time step, the animation pattern calls `pause`, which delays the program for some number of milliseconds (expressed here as the constant PAUSE_TIME). Without the call to `pause`, the program would finish faster than the human eye can follow.

The AnimatedSquare Program

```
public void run() {
    GRect square = new GRect(0, 0, SQUARE_SIZE, SQUARE_SIZE);
    square.setFilled(true);
    square.setFillColor(Color.RED);
    add(square);
    double dx = (getWidth() - SQUARE_SIZE) / N_STEPS;
    double dy = (getHeight() - SQUARE_SIZE) / N_STEPS;
    for (int i = 0; i < N_STEPS; i++) {
        square.move(dx, dy);
        pause(PAUSE_TIME);
    }
}
```