A Quick Overview of Methods

• At the most basic level, a method is a sequence of statements that has been collected together and given a name. The name makes it possible to execute the statements much more easily; instead of copying out the entire list of statements, you can just provide the method name.

• The following terms are useful when learning about methods:
  – Invoking a method using its name is known as calling that method.
  – The caller can pass information to a method by using arguments.
  – When a method completes its operation, it returns to its caller.
  – A method can pass information to the caller by returning a result.
Methods and Information Hiding

• One of the most important advantages of methods is that they make it possible for callers to ignore the inner workings of complex operations.

• When you use a method, it is more important to know what the method does than to understand exactly how it works. The underlying details are of interest only to the programmer who implements a method. Programmers who use a method as a tool can usually ignore the implementation altogether.

• The idea that callers should be insulated from the details of method operation is the principle of information hiding, which is one of the cornerstones of software engineering.
Methods as Tools for Programmers

- Particularly when you are first learning about programming, it is important to keep in mind that methods are not the same as application programs, even though both provide a service that hides the underlying complexity involved in computation.

- The key difference is that an application program provides a service to a user, who is typically not a programmer but rather someone who happens to be sitting in front of the computer. By contrast, a method provides a service to a programmer, who is typically creating some kind of application.

- This distinction is particularly important when you are trying to understand how the applications-level concepts of input and output differ from the programmer-level concepts of arguments and results. Methods like `nextInt` and `println` are used to communicate with the user and play no role in communicating information from one part of a program to another.
Method Calls as Expressions

- Syntactically, method calls in Java are part of the expression framework. Methods that return a value can be used as terms in an expression just like variables and constants.

- The Math class in the java.lang package defines several methods that are useful in writing mathematical expressions. Suppose, for example, that you need to compute the distance from the origin to the point \((x, y)\), which is given by

\[
\sqrt{x^2 + y^2}
\]

You can apply the square root function by calling the sqrt method in the Math class like this:

```java
double distance = Math.sqrt(x * x + y * y);
```

- Note that you need to include the name of the class along with the method name. Methods like Math.sqrt that belong to a class are called static methods.
## Useful Methods in the **Math** Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Math.abs(x)</code></td>
<td>Returns the absolute value of <code>x</code></td>
</tr>
<tr>
<td><code>Math.min(x, y)</code></td>
<td>Returns the smaller of <code>x</code> and <code>y</code></td>
</tr>
<tr>
<td><code>Math.max(x, y)</code></td>
<td>Returns the larger of <code>x</code> and <code>y</code></td>
</tr>
<tr>
<td><code>Math.sqrt(x)</code></td>
<td>Returns the square root of <code>x</code></td>
</tr>
<tr>
<td><code>Math.log(x)</code></td>
<td>Returns the natural logarithm of <code>x</code> (<code>\log_e x</code>)</td>
</tr>
<tr>
<td><code>Math.exp(x)</code></td>
<td>Returns the inverse logarithm of <code>x</code> (<code>e^x</code>)</td>
</tr>
<tr>
<td><code>Math.pow(x, y)</code></td>
<td>Returns the value of <code>x</code> raised to the <code>y</code> power (<code>x^y</code>)</td>
</tr>
<tr>
<td><code>Math.sin(theta)</code></td>
<td>Returns the sine of <code>theta</code>, measured in radians</td>
</tr>
<tr>
<td><code>Math.cos(theta)</code></td>
<td>Returns the cosine of <code>theta</code></td>
</tr>
<tr>
<td><code>Math.tan(theta)</code></td>
<td>Returns the tangent of <code>theta</code></td>
</tr>
<tr>
<td><code>Math.asin(x)</code></td>
<td>Returns the angle whose sine is <code>x</code></td>
</tr>
<tr>
<td><code>Math.acos(x)</code></td>
<td>Returns the angle whose cosine is <code>x</code></td>
</tr>
<tr>
<td><code>Math.atan(x)</code></td>
<td>Returns the angle whose tangent is <code>x</code></td>
</tr>
<tr>
<td><code>Math.toRadians(degrees)</code></td>
<td>Converts an angle from degrees to radians</td>
</tr>
<tr>
<td><code>Math.toDegrees(radians)</code></td>
<td>Converts an angle from radians to degrees</td>
</tr>
</tbody>
</table>
Method Calls as Messages

• In object-oriented languages like Java, the act of calling a method is often described in terms of sending a message to an object. For example, the method call

\[
\text{rect.setColor(Color.RED);}
\]

is regarded metaphorically as sending a message to the `rect` object asking it to change its color.

• The object to which a message is sent is called the receiver.
• The general pattern for sending a message to an object is

\[
\text{receiver\_name(arguments)}
\]
Writing Your Own Methods

- The general form of a method definition is

  ```java
  scope  type  name(argument list)  {
    statements in the method body
  }
  ```

  where `scope` indicates who has access to the method, `type` indicates what type of value the method returns, `name` is the name of the method, and `argument list` is a list of declarations for the variables used to hold the values of each argument.

- The most common value for `scope` is `private`, which means that the method is available only within its own class. If other classes need access to it, `scope` should be `public` instead.

- If a method does not return a value, `type` should be `void`. Such methods are sometimes called `procedures`. 
Returning Values from a Method

- You can return a value from a method by including a `return` statement, which is usually written as

  ```java
  return expression;
  ```

  where `expression` is a Java expression that specifies the value you want to return.

- As an example, the method definition

  ```java
  private double feetToInches(double feet) {
      return 12 * feet;
  }
  ```

  converts an argument indicating a distance in feet to the equivalent number of inches, relying on the fact that there are 12 inches in a foot.
Methods Involving Control Statements

• The body of a method can contain statements of any type, including control statements. As an example, the following method uses an if statement to find the larger of two values:

```java
private int max(int x, int y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```

• As this example makes clear, return statements can be used at any point in the method and may appear more than once.
The factorial Method

• The **factorial** of a number $n$ (which is usually written as $n!$ in mathematics) is defined to be the product of the integers from 1 up to $n$. Thus, $5!$ is equal to 120, which is $1 \times 2 \times 3 \times 4 \times 5$.

• The following method definition uses a **for** loop to compute the factorial function:

```java
private int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```
Nonnumeric Methods

Methods in Java can return values of any type. The following method, for example, returns the English name of the day of the week, given a number between 0 (Sunday) and 6 (Saturday):

```java
private String weekdayName(int day) {
    switch (day) {
    case 0: return "Sunday";
    case 1: return "Monday";
    case 2: return "Tuesday";
    case 3: return "Wednesday";
    case 4: return "Thursday";
    case 5: return "Friday";
    case 6: return "Saturday";
    default: return "Illegal weekday";
    }
}
```
Predicate Methods

• Methods that return Boolean values play an important role in programming and are called **predicate methods**.

• As an example, the following method returns **true** if the first argument is divisible by the second, and **false** otherwise:

```java
private boolean isDivisibleBy(int x, int y) {
    return x % y == 0;
}
```

• Once you have defined a predicate method, you can use it just like any other Boolean value. For example, you can print the integers between 1 and 100 that are divisible by 7 as follows:

```java
for (int i = 1; i <= 100; i++) {
    if (isDivisibleBy(i, 7)) {
        println(i);
    }
}
```
Using Predicate Methods Effectively

• New programmers often seem uncomfortable with Boolean values and end up writing ungainly code. For example, a beginner might write `isDivisibleBy` like this:

```java
private boolean isDivisibleBy(int x, int y) {
    if (x % y == 0) {
        return true;
    } else {
        return false;
    }
}
```

While this code is not technically incorrect, it is inelegant enough to deserve the bug symbol.

• A similar problem occurs when novices explicitly check to see if a predicate method returns `true`. You should be careful to avoid such redundant tests in your own programs.
Exercise: Testing Powers of Two

• Write a predicate method called `isPowerOfTwo` that takes an integer `n` and returns `true` if `n` is a power of two, and `false` otherwise. The powers of 2 are 1, 2, 4, 8, 16, 32, and so forth; numbers that are less than or equal to zero cannot be powers of two.

```java
private boolean isPowerOfTwo(int n) {
    if (n < 1) return false;
    while (n > 1) {
        if (n % 2 == 1) return false;
        n /= 2;
    }
    return true;
}
```
Mechanics of the Method-Calling Process

When you invoke a method, the following actions occur:

1. Java evaluates the argument expressions in the context of the calling method.

2. Java then copies each argument value into the corresponding parameter variable, which is allocated in a newly assigned region of memory called a stack frame. This assignment follows the order in which the arguments appear: the first argument is copied into the first parameter variable, and so on.

3. Java then evaluates the statements in the method body, using the new stack frame to look up the values of local variables.

4. When Java encounters a return statement, it computes the return value and substitutes that value in place of the call.

5. Java then discards the stack frame for the called method and returns to the caller, continuing from where it left off.
The Combinations Function

• To illustrate method calls, the text uses a function $C(n, k)$ that computes the combinations function, which is the number of ways one can select $k$ elements from a set of $n$ objects.

• Suppose, for example, that you have a set of five coins: a penny, a nickel, a dime, a quarter, and a dollar:

How many ways are there to select two coins?

- penny + nickel
- penny + dime
- penny + quarter
- penny + dollar
- nickel + dime
- nickel + quarter
- nickel + dollar
- dime + quarter
- dime + dollar
- quarter + dollar

for a total of 10 ways.
Combinations and Factorials

• Fortunately, mathematics provides an easier way to compute the combinations function than by counting all the ways. The value of the combinations function is given by the formula

\[
C(n,k) = \frac{n!}{k! \times (n-k)!}
\]

• Given that you already have a `factorial` method, it is easy to turn this formula directly into a Java method, as follows:

```java
private int combinations(int n, int k) {
    return factorial(n) / (factorial(k) * factorial(n - k));
}
```

• The next slide simulates the operation of `combinations` and `factorial` in the context of a simple `run` method.
The Combinations Program

```java
public void run() {
    int n = readInt("Enter number of objects in the set (n): ");
    int k = readInt("Enter number to be chosen (k): ");
    println("C(" + n + ", " + k + ") = " + combinations(n, k));
}
```

```
public int combinations(int n, int k) {
    return factorial(n) / (factorial(k) * factorial(n - k));
}
```

```
private int factorial(int n) {
    int result = 1;
    for ( int i = 1 ; i <= n ; i++ ) {
        result *= i;
    }
    return result;
}
```

```
Enter number of objects in the set (n): 5
Enter number to be chosen (k): 2
C(5, 2) = 10
```
Decomposition

One of the most important advantages of methods is that they make it possible to break a large task down into successively simpler pieces. This process is called decomposition.

Once you have completed the decomposition, you can then write a method to implement each subtask.
Controlling Access to Entries

- Each entry in a Java class is marked with one of the following keywords to control which classes have access to that entry:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>All classes in the program have access to any public entry. The public entries in a class are said to be <strong>exported</strong> by that class.</td>
</tr>
<tr>
<td>private</td>
<td>Access to entries declared as private is limited to the class itself, making that entry completely invisible outside the class.</td>
</tr>
<tr>
<td>protected</td>
<td>Protected entries are restricted to the class that defines them, along with any of its subclasses or any classes in the same package.</td>
</tr>
<tr>
<td>(no keyword)</td>
<td>If the access keyword is missing, the entry is visible only to classes in the same package. Such entries are called <strong>package-private</strong>.</td>
</tr>
</tbody>
</table>

- The text uses only **public** and **private**. All entries are marked as **private** unless there is a compelling reason to export them.
Representing Student Information

• Understanding the structure of a class is easiest in the context of a specific example. The next four slides walk through the definition of a class called **Student**, which is used to keep track of the following information about a student:
  – The name of the student
  – The student’s six-digit identification number
  – The number of credits the student has earned (which may include a decimal fraction to account for half- and quarter-credit courses)
  – A flag indicating whether the student has paid all university fees

• Each of these values is stored in an instance variable of the appropriate type.

• In keeping with the modern object-oriented convention used throughout both the book and the ACM Java Libraries, these instance variables are declared as **private**. All access to these values is therefore mediated by methods exported by the **Student** class.
The Student Class

The Student class keeps track of the following pieces of data about a student: the student's name, ID number, the number of credits the student has earned toward graduation, and whether the student is paid up with respect to university bills. All of this information is entirely private to the class. Clients can obtain this information only by using the various methods defined by the class.

```java
public class Student {

/**
 * Creates a new Student object with the specified name and ID.
 * @param name The student's name as a String
 * @param id The student's ID number as an int
 */

public Student(String name, int id) {
    studentName = name;
    studentID = id;
}

This comment describes the class as a whole.

This comment describes the constructor.

The class header defines Student as a direct subclass of Object.

The constructor sets the instance variables.
The Student Class

/**
 * Creates a new Student object with the specified name and ID.
 * @param name The student's name as a String
 * @param id The student's ID number as an int
 */
public Student(String name, int id) {
    studentName = name;
    studentID = id;
}

/***
 * Gets the name of this student.
 * @return The name of this student
 */
public String getName() {
    return studentName;
}

/***
 * Gets the ID number of this student.
 * @return The ID number of this student
 */
public int getID() {
    return studentID;
}

/***
 * Sets the number of credits earned.
 * @param credits The new number of credits earned
 */
public void setCredits(double credits) {
    creditsEarned = credits;
}
The Student Class

/**
 * Gets the name of this student.
 * @return The name of this student
 */
public String getName() {
    return studentName;
}

/**
 * Gets the ID number of this student.
 * @return The ID number of this student
 */
public int getID() {
    return studentID;
}

/**
 * Sets the number of credits earned.
 * @param credits The new number of credits earned
 */
public void setCredits(double credits) {
    creditsEarned = credits;
}

/**
 * Gets the number of credits earned.
 * @return The number of credits this student has earned
 */
public double getCredits() {
    return creditsEarned;
}

/**
 * Sets whether the student is paid up.
 * @param flag The value true or false indicating paid-up status
 */
public void setPaidUp(boolean flag) {
    paidUp = flag;
}

/**
 * Returns whether the student is paid up.
 * @return Whether the student is paid up
 */
public boolean isPaidUp() {
    return paidUp;
}

Names for getter methods usually begin with the prefix get. The only exception is for getter methods that return a boolean, in which case the name typically begins with is.
The Student Class

/**
 * Creates a string identifying this student.
 * @return The string used to display this student
 */
public String toString() {
    return studentName + " (#" + studentID + ")";
}

/* Public constants */
/** The number of credits required for graduation */
public static final double CREDITS_TO_GRADUATE = 34.0;

/* Private instance variables */
private String studentName;   /* The student's name */
private int studentID;        /* The student's ID number */
private double creditsEarned; /* The number of credits earned */
private boolean paidUp;       /* Whether student is paid up */
Using the Student Class

• Once you have defined the Student class, you can then use its constructor to create instances of that class. For example, you could use the following code to create two Student objects:

```java
Student chosenOne = new Student("Harry Potter", 123456);
Student topStudent = new Student("Hermione Granger", 314159);
```

• You can then use the standard receiver syntax to call methods on these objects. For example, you could set Hermione’s number-of-credits field to 97 by writing

```java
topStudent.setCredits(97);
```

or get Harry’s full name by calling

```java
chosenOne.getName();
```
Exercise: Design an Employee Class

• Create a definition for a class called Employee, which keeps track of the following information:
  – The name of the employee
  – A number indicating the order in which this employee was hired
  – A flag indicating whether the employee is still active
  – The salary (a number that may contain a decimal fraction)

• The name and employee number should be assigned as part of the constructor call, and it should not be possible to change them subsequently. By default, new employees should be marked as active. The salary field need not be initialized.

• The class should export appropriately named getters for all four fields and setters for the last two.
The Employee Class

/**
 * The Employee class keeps track of the following pieces of
 * data about an employee: the name, employee number, whether
 * the employee is active, and the annual salary.
 */

public class Employee {

/**
 * Creates a new Employee object with the specified name and
 * employee number.
 * @param name The employee's name as a String
 * @param id The employee number as an int
 */
    public Employee(String name, int id) {
        employeeName = name;
        employeeNumber = id;
        active = true;
    }
The Employee Class

/**
 * The Employee class keeps track of the following pieces of
 * data about an employee: the name, employee number, whether
 * the employee is active, and the annual salary.
 */

public class Employee {

/**
 * Creates a new Employee object with the specified name and
 * employee number.
 * @param name The employee's name as a String
 * @param id The employee number as an int
 */
    public Employee(String name, int id) {
    employeeName = name;
    employeeNumber = id;
    active = true;
    }

/**
 * Gets the name of this employee.
 * @return The name of this employee
 */
    public String getName() {
    return employeeName;
    }

/**
 * Gets the employee number of this employee.
 * @return The employee number of this employee
 */
    public int getEmployeeNumber() {
    return employeeNumber;
    }

/**
 * Sets whether the employee is active.
 * @param flag The value true or false indicating active status
 */
    public void setActive(boolean flag) {
    active = flag;
    }
}
The Employee Class

```java
/**
 * Gets the name of this employee.
 * @return The name of this employee
 */
public String getName() {
    return employeeName;
}

/**
 * Gets the employee number of this employee.
 * @return The employee number of this employee
 */
public int getEmployeeNumber() {
    return employeeNumber;
}

/**
 * Sets whether the employee is active.
 * @param flag The value true or false indicating active status
 */
public void setActive(boolean flag) {
    active = flag;
}

/**
 * Returns whether the employee is active.
 * @return Whether the employee is active
 */
public boolean isActive() {
    return active;
}

/**
 * Sets the employee's salary.
 * @param salary The new salary
 */
public void setSalary(double salary) {
    annualSalary = salary;
}

/**
 * Gets the annual salary for this employee.
 * @return The annual salary for this employee
 */
public double getSalary() {
    return annualSalary;
}
```
The Employee Class

/**
 * Creates a string identifying this employee.
 * @return The string used to display this employee
 */
    public String toString() {
        return employeeName + " (#" + employeeNumber + ")";
    }

/* Private instance variables */
private String employeeName; /* The employee's name */
private int employeeNumber; /* The employee number */
private boolean active; /* Whether the employee is active */
private double annualSalary; /* The annual salary */
}
Exercise: Using the Employee Class

• Now that you have defined Employee, write declarations for three variables that contain the names of the following three employees: Ebenezer Scrooge (employee #1), Jacob Marley (employee #2), and Bob Cratchit (employee #3).

```java
Employee founder = new Employee("Ebenezer Scrooge", 1);
Employee partner = new Employee("Jacob Marley", 2);
Employee clerk = new Employee("Bob Cratchit", 3);
```

• Using these variables, write a Java statement that marks the Employee instance for Jacob Marley as inactive.

```java
partner.setActive(false);
```

• Write a Java statement that doubles Bob Cratchit’s salary.

```java
clerk.setSalary(2 * clerk.getSalary());
```
The End