COMP-202
Unit 5: Methods

Introduction (1)

• So far, all the programs we wrote consisted of writing a single method called `main()` within a single class
  – All the statements we wrote were part of the body of this method
• However, some parts of certain programs we wrote are useful in many situations; these program parts often consist of more than one statement
  – For example, a previous exercise involved writing a program which displayed and counted all prime numbers between 2 and a value entered by the user
  – This involved processing each number in this range one by one to verify whether it was prime or not
  – The portion of the program that verified whether a number was prime or not consisted of more than a single statement

Introduction (2)

• However, verifying that a number is prime is useful in many situations; lots of programs which do different things need to check if some numbers are prime
• How can we reuse a useful list of statements in more than one program?
• One way to do so would be to copy this list of statements and paste it in all programs in which it is needed; however, this has very serious disadvantages:
  – We may have to modify the copies of the original list of statements to ensure that they work properly with the programs in which the copies are pasted
  – If the original list of statements contains a mistake, then we have to fix this mistake in all the programs which contain a copy of this list

Methods (1)

• In Java, a `method` is a list of statements that has been given a name
  – The statements that the method consists of perform a task when they are executed together as a unit
  – In other programming languages, methods are called `procedures` or `functions`
• Writing a method means specifying a list of statements and giving a name to this list of statements
• Once a method is written, its name can be used as a single instruction inside another method
  – The execution of that instruction results in the execution of all statements which are part of the method
  – This is referred to as calling or invoking a method
Methods (2)

- We use the terms method call or simply a call to refer to the instruction which cause the statements that are part of a method to be executed.
- For every method call:
  - The method which contains and issues this call is often referred to as the calling method, the caller method, or simply the caller.
  - The method whose statements are executed as a result of the call is often referred to as the called method.

Part 1: Defining Methods

Java Program Structure Revisited

- Recall the basic structure of programs in Java:
  - A program consists of one or more classes; each class is usually defined in a different file.
  - A class contains one or more methods.
  - A method contains program statements.
- Until now, all programs we have developed involved writing only one class, and this class contained exactly one method: `main()`.
  - This is about to change.

Method Structure (1)

- A method declaration specifies the code that will be executed when the method is invoked (or called).
- A method declaration consists of two parts: the method header and the method body.
- The header of a method specifies important characteristics of the method, such as its name.
- The body of a method contains the actual statements that are part of the method.
  - The body of a method is delimited by braces `{ }`.
- In Java, all method declarations are part of the body of a class.
  - Methods cannot be declared outside of a class or inside other methods.
Method Structure (2)

```java
public class MyClass {
    public static void myMethod() {
        // Body of method myMethod()
    }
    public static void myOtherMethod() {
        // Body of method myOtherMethod()
    }
}
```

Simple Method Declaration

- Below is a declaration for a simple method which calculates the sum of two integers:
  ```java
  public static int add(int n1, int n2) {
      int sum;
      sum = n1 + n2;
      return sum;
  }
  ```

Method Header

- A method declaration begins with a method header; it contains important information about the method, such as:
  - The method's name
  - What the method takes as input
  - What the method produces as output

- Below is the header of the simple `add()` method:
  ```java
  public static int add(int n1, int n2)
  ```

General Method Header Syntax

```
modifiers returnType methodName (parameterList)
```

Where:

- `modifiers` - public, private, static, ...
  (there can be more than one) (optional)
- `returnType` - primitive or reference type
  (mandatory)
- `methodName` - identifier, as defined previously
  (mandatory)
  ```java
  (parameterList)
  ```
  `parameterList` - defined later (optional)
  ```java
  )
  ```

Note the absence of semi-colon after the method header
Access Modifiers (1)

- Method modifiers are used to change some of the method's properties
- One of these properties determines which methods can call the method being declared
- Modifiers which change this property are called access modifiers (or visibility modifiers)
- There are four access modifiers in Java, but only two are important in this course
  - When a method is private, only the methods declared in the same class can call it; that is, if a private method \( m() \) is declared in a class called \( C \), then only the methods defined in class \( C \) can call method \( m() \)

Access Modifiers (2)

- On the other hand, when a method is public, any method declared in any class can call it
- Access modifiers are mutually exclusive; that is, a method header cannot specify more than one access modifier
- Note that forgetting to specify an access modifier will result in a default access modifier being used; this default modifier is neither public nor private

private vs. public Methods

- Methods that are to be publicly accessible should be public
- However, we might have some helper methods used for internal decomposition
  - These methods support other methods in the class (other methods in the class call them), but they should not be accessible by methods outside the class
  - Therefore, they should be declared private

Other Method Modifiers

- Another important modifier is static
  - For now, all the methods we write will be declared static
  - Therefore, the static modifier MUST be included in the header of each method that we write (at least for now)
  - We will cover non-static methods (also called instance methods) when we cover objects
Method Input

- Many methods require input
- The input variables of a method are called its *parameters* (or sometimes its *arguments*)
  - When the method is called, the input values are copied into the method's parameters
  - The method does not care how the values copied in its parameters were originally produced by its caller; they could have been read from the keyboard, or they could be the result of other calculations
- A method can have zero, one, or many parameters
- Each parameter is of a certain type, but this type can be anything (primitive or reference)

Method Parameter Lists (1)

- The *parameter list* specifies the type and name of each parameter that a method accepts
  - The parameter list is optional; if it is absent, then the method accepts no parameters
  - The name of a parameter in the method declaration is called a *formal parameter*
  - Formal parameters must be valid Java identifiers, just like variable names
  - In the parameter list, each parameter declaration **MUST** consist of the type of the parameter, followed by the formal parameter
  - Parameter declarations are separated by commas

Method Parameter Lists (2)

- Examples of parameter lists:
  - `public static int myMethod()` // Valid; accepts no parameters
  - `public static int myMethod(int i)` // Valid; accepts one parameter of type int
  - `public static int myMethod(int i1, int i2, double d)` // Valid; accepts three parameters, two of type int // followed by one of type double
  - `public static int myMethod(int i1, i2, double d)` // Invalid; no type specified for i2
  - `public static int myMethod(int i1, double d, )` // Invalid; comma after last parameter declaration

Method Output

- Many methods produce output
- The output of a method is called its *return value*
  - When the called method terminates its execution, the return value is passed back to the caller
  - The called method does not care how the caller uses the return value is used after the latter is produced; the return value could be displayed, used in other calculations, or even discarded
- The return value is of a certain type, called the *return type*; the return type can also be anything (primitive or reference)
- A method can only return one value
  - That is, a method cannot return two values at the same time, regardless of whether the types of these values are the same or different
**void (1)**

- In Java, a method does not have to return a value to its caller
  - Some methods do not return anything
- The return type of methods which do not return anything is **void**
- Consider the following method declaration:
  ```java
  public static void main(String[] args) {
  }
  ```
  Specifies that the method does not return anything to its caller
- The fact that a method does not return a value and has return type **void** does not mean that the method does nothing

**void (2)**

- The statements in the body of a method whose return type is **void** are executed normally when the method is called
- The statements in the `main()` method are executed even though its return type is **void**

**Method Signatures**

- The compiler identifies a method by its *signature*
- A method signature is defined to be:
  - The method's name
  - The method's parameters, their types and the order in which they are listed
- Different method signatures:
  - `public static void myMethod()`
  - `public static void myMethod(int i)`
  - `public static void myMethod(double d)`
  - `public static void myMethod(int i, double d)`
  - `public static void myMethod(double d, int i)`
- Note that the return type is *not* part of the method signature

**Method Input / Output**

- Whether or not a method requires parameters (and the number of parameters it requires if it does require any) is independent of whether or not it returns a value
  - Some methods do not require parameters, and do not return a value
  - Some methods require parameters, and do not return a value
  - Some methods do not require parameters, and return a value
  - Some methods require parameters, and return a value
Method Body (1)

- The method header is followed by the method body between curly braces ({})
- The method body contains the statements which will be executed when the method is called

```java
public static int add(int n1, int n2) {
    int sum;
    sum = n1 + n2;
    return sum;
}
```

Statements to be executed when method `add()` is called

Method Body (2)

- The body of a method can contain statements of any type that we have seen so far (and more):
  - Variable declarations
  - Assignment statements
  - Conditional statements
  - Loops
  - Calls to (probably other) methods

Formal Parameter Usage (1)

- Formal parameters can be used in the method body the same way regular variables are used
- Notice how the following method uses the formal parameters to calculate the sum of the values it receives as input:

```java
public static int add(int n1, int n2) {
    int sum;
    sum = n1 + n2;
    return sum;
}
```

- When the `add()` method is called, a value is copied in each of the two formal parameters `n1` and `n2`
  - This happens automatically; we will cover how the parameter passing mechanism works in detail later

Formal Parameter Usage (2)

- Therefore, after the statement `sum = n1 + n2;` is executed, the value stored in variable `sum` will be the sum of whatever values were copied in formal parameters `n1` and `n2` when the method was called, regardless of what these values were, or how they were originally computed
Local Variables

- A method can declare its own variables
- These variables are *local* to the method, and therefore called *local variables*
- Local variables are created (that is, the memory is allocated) each time the method is called; they are discarded when the method finishes execution
- Therefore, a local variable \( v \) declared in method \( m() \) does not retain its value between invocations of method \( m() \)

The return Statement (1)

- A *return statement* terminates the method
  - It can occur anywhere in a method, and when it is executed, control goes back to the calling method
  - There can be more than one *return statement* in a method
  - If the method's return type is not *void*, the return statement specifies the value that will be returned as return value
- If a method does not have a *void* return type, all execution paths through this method **MUST** end with a *return statement*
- Methods whose return types are *void* are not required to have any *return statements*

The return Statement (2)

- Consider the following method:
  ```java
  public static int absoluteValue(int i) {
      if (i < 0) {
          return -i;
      }
  }
  ```
- The above causes an error, because if \( i < 0 \) evaluates to \( false \), the execution of the method ends without executing a *return statement*
- Making the *return statement* the last statement of the method ensures that all execution paths end with a *return statement*

The return Statement (3)

- If a method's return type is not *void*, all *return statements* in this method must include an expression
  - This expression is evaluated, and the value is returned to the method's caller as return value
- The type of the value the expression in the *return statement* evaluates to **MUST** be either:
  - of the same type as the method's return type
  - a type which can be converted to the return type using an assignment conversion
- A *return statement* in a method whose return type is *void* cannot include any expressions
Part 2: Calling / Invoking Methods

Calling Methods (1)

- A method call has the following syntax:

```
className . methodName( parameters )
```

- The name of the class in which the method is declared
- The name of the method being called
- A list of expressions separated by commas; there MUST be as many expressions as the number of parameters the method accepts
- These expressions are evaluated before the call, and the values are passed to the method being called as parameters

Calling Methods (2)

- Additional syntax rules of calling methods:
  - If the calling method and the called method are declared in the same class, the class name and the dot (.) in front of the method name are optional
  - If the method call is the only component of a statement, it must be followed by a semi-colon

Calling Methods (3)

- Note that the person who writes a method decides:
  - what the method will be called
  - how many parameters the method will accept
  - what the types of these parameters will be
  - the type of the value that the method will return
- Therefore, doing any of the following will result in the compiler reporting errors:
  - Attempting to call a method using a different name
  - Passing a number of parameters which is different from the number of parameters the method accepts
  - Passing parameters whose types are incompatible with the types of the method's formal parameters
Method Control Flow (1)

- When a method is called, the flow of control jumps to the beginning of the called method and starts executing the statement it contains.
- When the method's execution is complete, the flow of control returns to the point in the caller where the method call is, and continues from there.

Method Control Flow (2)

Parameter Passing (1)

- As we know, when a method is declared, its designers specify how many parameters the method accepts, as well as the type of each of these parameters.
- When we call a method, we must supply actual values as parameters to the method so that it can do what it has been designed to do.
  - These values are called the actual parameters.
- Each time a method is called, the values of the actual parameters in the method call are copied into the formal parameters of the method.

Parameter Passing (2)

- The value of the first actual parameter is copied in the first formal parameter in the method's parameter list, the value of the second actual parameter is copied in the second formal parameter in the method's parameter list, and so on.
Parameter Passing (3)

public class TestAdd {
    public static void main(String[] args) {
        int number = 3;
        int theSum = TestSum.add(number, 1);
    }
}

public static int add(int n1, int n2) {
    int sum;
    sum = n1 + n2;
    return sum;
}

Parameter Passing Details (1)

• When passing parameters to a method, the passing mechanism works like an assignment statement
  – The actual parameter does not have to be a variable; you can pass any expression as an actual parameter when invoking a method
• Each actual parameter is like the right side of the assignment statement
  – Formal parameters MUST be variables, they cannot be expressions

Parameter Passing Details (2)

• The expression used as actual parameter will be evaluated before the method is called
  – The value the expression evaluates to will be copied in the formal parameter before the statements in the called method called start to be executed
• The type of the value the expression in the actual parameter evaluates to MUST be either
  – a type which can be converted to the formal parameter's type using an assignment conversion
  – of the same type as the formal parameter
• This type of parameter passing mechanism is called passing by value

Passing Primitive Types (1)

• A consequence of the fact that parameters are passed by value is that changing the value of the formal parameter inside the method does not affect the value of the actual parameter
• Suppose that the method declaration is
  public static void m(int i, int j) {
      /* Code */
  }
  and the call to method m() (in, say, method main()) is
  m(a, 2 * b + c);
• Then, when method m() is called, something like this occurs automatically to pass the parameters:
  \[ i_m = a_{main} \]
  \[ j_m = 2 \times b_{main} + c_{main} \]
Passing Primitive Types (2)

In other words:

– The value of the variable declared in method `main()` is copied in the formal parameter declared in the header of method `m()`.

– The expression `2 * b + c`, where `b` and `c` are variables declared in method `main()`, is evaluated; the value this expression evaluates to is copied in the formal parameter declared in the header of method `m()`.

Therefore, changing the value of formal parameter inside method `m()` does not change the original value stored in the variable declared in the `main()` method.

public class PrimitiveAssignmentDemo {
    public static void main(String[] args) {
        int i, j;
        i = 42;
        System.out.println("Value of i before assignment: "+ i);
        j = i;
        System.out.println("Value assigned to j: "+ j);
        j = 0;
        System.out.println("New value of j: "+ j);
        System.out.println("Value of i: "+ i);
    }
}

What does this display?

m()
i: 42
j: 0

// Continued from previous slide

public class PrimitivePassingDemo {
    public static void main(String[] args) {
        int i;
        i = 42;
        System.out.println("i before calling m() == "+ i);
        System.out.println("--");
        m(i);
        System.out.println("-"+ i);
        System.out.println("i after calling m() == "+ i);
    }
    // Continued on next slide
}

What does this display?

m()
i: 42
j: 0

// Continued from previous slide

public static void m(int j) {
    System.out.println("Received j == "+ j);
    j = 0;
    System.out.println("New j == "+ j);
}

What does this display?

m()
i: 42
j: 0

What does this display?
Method Calls and Statements

- A call to a method whose return type is **void** is a statement in itself
  - A statement containing a call to a method whose return type is **void** **MUST NOT** consist of anything except the method call
  - The method call **MUST** be followed by a semi-colon

Method Calls and Expressions (1)

- On the other hand, a call to a method which returns a value can be used as an operand in a larger expression
  - When evaluating the expression, the value returned by the method is used as the operand in the expression

- Examples:
  - int s = add(a, b);
    Assigns the value returned by the call to method `add()` with actual parameters `a` and `b` to variable `s`
  - int s = add(a, b) / add(c, d);
    Divides the value returned by the call to method `add()` with actual parameters `a` and `b` by the value returned by the call to method `add()` with actual parameters `c` and `d`; assigns the quotient to variable `s`

Method Calls and Expressions (2)

- int s = add(a, add(b, c));
  Passes the value returned by a call to method `add()` with actual parameters `b` and `c` to another call to method `add()` as an actual parameter, along with actual parameter `a`; assigns the value of this second call to variable `d`

- if (add(a, b) > s) { /* code */ }
  Compares the value returned by the call to method `add()` with actual parameters `a` and `b` to the value of variable `s`; if the value returned by the method call is greater than the value of variable `s`, then the statements in the if-statement are executed, otherwise they are skipped

Part 3: Dependencies and Libraries
Dependencies

- Some classes are stand-alone; they do not depend on any other class to be available in order to work (at least not at first glance)
  - They do not call methods defined in other classes
  - They do not use other classes in any other way
- However, when a class $C_1$ calls methods defined in class $C_2$, then we say that class $C_1$ depends on $C_2$, or that a dependency exists between $C_1$ and $C_2$

Dependency Problems

- If you try to compile a class which depends on some other class, and the Java compiler cannot find this other class, the compiler will report errors
- Likewise, if you run a program involving a class which depends on another class, and the Java Virtual Machine cannot find this second class when it is needed, your program will crash
  - When this occurs, the error message will mention that the program threw a `NoClassDefFoundError`
  - The error message will mention which line in your program caused the latter to crash; it may also mention which class could not be found (and therefore caused the crash)

Classpath (1)

- In all Java environments, there is a setting which tells the compiler and the virtual machine where to look for classes when trying to resolve dependencies
  - This setting is called the `Classpath`
- The `Classpath` is a list of locations
  - Each of these locations can be a directory or an archive file
- When trying to compile a class which depends on another class, the compiler will look for this second class in all locations listed in the Classpath
  - If this other class cannot be found in any of the locations listed in the Classpath, then the compiler will produce errors

Classpath (2)

- Likewise, when running a program involving a class which depends on another class, the Java Virtual Machine will look for this second class in all locations listed in the Classpath
  - If this other class cannot be found in any of the locations listed in the Classpath, then the program will crash
- In most Java environments, the Classpath will be configured automatically so that when you compile a `.java` file or run a `.class` file, the directory which contains the file is on the Classpath
  - This means that if you put two `.java` files in the same directory, the class defined in one of the files will be able to use the class defined in the other file, without doing anything special
  - Classpath configuration is generally beyond the scope of this course
Libraries

- Recall: One of the roles that a class can have is to be a collection of related methods
- A class which has this role is sometimes called a library
- Methods which are used by many programs should be placed in libraries
  - That way, these methods can be maintained and updated separately from the programs in which they are used

Class Libraries

- A *class library* is a collection of classes that we can use when developing programs
- There is a *Java standard class library* that is part of any Java development environment
- These classes are not part of the Java language per se, but we rely on them heavily
- Other class libraries can be obtained through third party vendors, or you can create them yourself

The Math Class (1)

- The Math class contains many methods which providing various mathematical functions, such as absolute value, trigonometric functions, square root, power, logarithms, and others
- The Math class is perhaps the best example of a Java class whose role is to be a collection of related methods
- Some of the methods declared in the Math class:
  - public static double sin(double a): Returns the trigonometric sine of angle $a$, where $a$ is expressed in radians
  - public static double cos(double a): Returns the trigonometric cosine of angle $a$, where $a$ is expressed in radians
  - public static double tan(double a): Returns the trigonometric tangent of angle $a$, where $a$ is expressed in radians

The Math Class (2)

- public static double toRadians(double angrad): Converts angle $angrad$ measured in radians to an approximately equivalent angle measured in degrees
- public static double toDegrees(double angdeg): Converts angle $angdeg$ measured in degrees to an approximately equivalent angle measured in radians
- public static double pow(double a, double b): Returns the value of $a$ raised to the power of $b$
- public static double sqrt(double a): Returns the positive square root of $a$
- public static double exp(double a): Returns Euler's number $e$ raised to the power of $a$
- public static double log(double a): Returns the natural logarithm (base $e$) of $a$
The Math Class (3)

- public static double log10(double a): Returns the base 10 logarithm of a
- public static double random(): Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

Math Class: Exercise

- Complete the main() method of the LawOfCosines class by adding code that does the following:
  - Computes the length of the third side of a triangle from the other two sides and the angle between them; the formula for this calculation is \( c^2 = a^2 + b^2 - 2ab \cdot \cos \theta \), where \( c \) is the side whose length is unknown, \( a \) and \( b \) are the two sides whose lengths are known, and \( \theta \) is the angle between sides \( a \) and \( b \)
  - Displays the result with an appropriate message
- Note that the parameter passed to the cos() method of the Math class is an angle expressed in radians, while the program asks for an angle expressed in degrees; you will therefore need to convert the angle from degrees to radians

LawOfCosines.java

```java
import java.util.Scanner;

public class LawOfCosines {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        double a, b, theta;

        System.out.print("Enter the first side of the triangle: ");
        a = keyboard.nextDouble();
        System.out.print("Enter the second side of the triangle: ");
        b = keyboard.nextDouble();
        System.out.print("Enter the angle between the two sides " +
                       "(in degrees): ");
        theta = keyboard.nextDouble();

        // Add your code here
    }
}
```

Part 4: Scope
Scope (1)

- The scope of a variable is the area in a program in which that variable can be used
  - It also determines when the variable is created and when it is destroyed
- The location of a variable declaration establishes its scope, and places it in one of three categories
  - Formal parameters are variables which are declared in the header of a method
  - Local variables are variables which are declared anywhere inside the body of a method
  - Member variables are variables which are declared in the class body, outside of all methods; we will cover those, as well as their scoping rules, later in the course

Scope (2)

- Consider the following class declaration:

```java
public class MyClass {
    public static void myMethod(double myDouble) {
        boolean myBoolean;
        /* Method code */
    }
}
```

Scope of Local Variables

- Local variables are declared inside the body of a method
  - The scope of a local variable extends from its declaration to the end of the code block which contains this declaration
  - The end of this code block is defined by the closing curly brace (`}`)
- Formal parameters are declared in method headers, and are used to pass values to methods
  - The scope of a formal parameter is the entire method for which it is a parameter
  - It is treated just like a local variable
- It is impossible to access the formal parameters and local variables of a method from inside any other method

Nested Scopes

- Every code block delimited by curly braces (`{}`) delimits the scope of the variables declared in it
- Inside a code block, the following variables are visible:
  - All variables defined within that block
  - All variables defined in all containing blocks
  - In other words, if a block is contained inside another block, the inside block can see all variables that the outside block can see in addition to the ones it defines
  - All other variables are invisible
Nested Scopes Example

```java
public class ScopeDemo {
  public static void myMethod(int a) {
    int b;
    if (a == 0) {
      int c;
      } else {
      int d;
      if (a > 100) {
        int e;
        }
      }
    }
  }
}
```

Scope Errors

- Consider the following code fragment:
  ```java
  int i = keyboard.nextInt();
  if (i < 0) {
    int j = -i;
  }
  System.out.println(j);
  ```
- The final line will cause a compilation error:
  - The scope of variable `j` extends only from its declaration to the end of the block statement it is declared in
  - Because the call to `println()` is outside the scope of variable `j`, it causes an error
  - Solution: Either move the `println()` call inside the `if` clause of the `if` statement, or move the declaration of `j` before the `if` statement

Scoping Rules (1)

- In general, we cannot declare a variable in a block if it has the same name as another variable declared in the same block, or in any containing block
- Implications:
  - Two parameters to the same method cannot have the same name
  - Two parameters to different methods can have the same name
  - A local variable cannot have the same name as a parameter
  - Two local variables declared in the same method cannot have the same name, unless the variables are declared in different blocks within the method, and neither of these blocks contains the other
  - Two local variables declared in different methods can have the same name

Scoping Rules (2)

- Note that two parameters which have the same name but are parameters to different methods are completely unrelated
- Likewise, two local variables which have the same name but are declared in different methods are completely unrelated
- Finally, local variables and formal parameters which have the same name are completely unrelated if they are declared in different methods
**Scope and the for Loop**

- If you declare the index variable for a for loop inside the initialization clause, its scope extends only to the for loop (header and body).
- In such cases, if you try to use the index variable outside the loop body, the compiler will generate an error:
  ```java
  for (int i = 0; i < 5; i++) {
      System.out.println(i);
  }
  System.out.println(i);
  // Error: variable i is only defined within the loop body
  ```

**Scope Advice**

- Some advice to avoid scope-related errors:
  - Declare all local variables at the beginning of the methods in which they are declared (possible exception: loop indices used with for loops)

**Part 5: Method Overloading**

- **Method overloading** is the process of using the same method name for multiple methods declared in the same class.
- The signature of each overloaded method must be unique.
- Recall: the signature of a method consists of:
  - the method's name
  - the number of parameters it accepts
  - the types of the parameters it accepts
  - the order of the parameters' types
- Also recall: the return type of the method is *not* part of this signature.
Method Overloading (2)

- When a method is called, the compiler must be able to determine which version of the method is being invoked by analyzing the types of the actual parameters
  - If an ambiguity exists, the compiler will report an error

Overloading Example (1)

- Consider the following class definition:

  ```java
  public class OverloadingDemo {
      public static double tryMe(int x) {
          return x + 0.375;
      }
      public static double tryMe(int x, double y) {
          return x * y;
      }
      public static void main(String[] args) {
          double result = tryMe(25, 4.32);
          System.out.println(result);
      }
  }
  ```

- What will be displayed when the `main()` method is executed?
  - Answer: 108.0
  - The call to method `tryMe()` in method `main()` passes two actual parameters; the first is of type `int`, the second is of type `double`
  - Therefore, the version of method `tryMe()` which accepts a parameter of type `int` and one of type `double` (in that order) is invoked, not the version which takes a single parameter of type `int`

Overloading Example (2)

- What will be displayed when the `main()` method is executed?
  - Answer: 108.0
  - The call to method `tryMe()` in method `main()` passes two actual parameters; the first is of type `int`, the second is of type `double`
  - Therefore, the version of method `tryMe()` which accepts a parameter of type `int` and one of type `double` (in that order) is invoked, not the version which takes a single parameter of type `int`

More Overloading Examples

- Many methods defined in the Java Standard Class Library are overloaded
  - The `print()` and `println()` methods defined in class `PrintStream` (System.out is variable of type `PrintStream`) are overloaded:
    - `print(String s) / println(String s)`
    - `print(int i) / println(int i)`
    - `print(double d) / println(double d)`
  - Assuming that `total` is a variable whose type is not `String`, The following lines invoke different versions of the `println()` method:
    - `System.out.println("The total is: ");`
    - `System.out.println(total);`
Part 6: Advantages of Methods

Advantages of Methods (1)

• Methods are an elegant way to reuse useful lists of statements
  – They remove the need to copy and paste statements
  – If the list of statements contains a mistake, we only need to fix this mistake in one place
  – Likewise, if we want to replace the list of statements by one which performs the same task more efficiently, we only need to change the statements in one place
  – Moreover, they enable us to have the programs we write use the lists of statements someone else has written; this has the potential to greatly reduce our workload, as someone else has already written parts of the program for us

Advantages of Methods (2)

• Methods make it easier to write large, complex programs
  – If we need to write a program which performs a complex task, it does not need to be a single large program
  – Instead, we can decompose the complex task into many simpler subtasks, write a method to perform each subtask, and have the program call these methods

Advantages of Methods (3)

• Methods allow us to keep details abstract
  – A method is abstract in that we don't really have to think about its internal details in order to use it, just like we don't really have to think about how the engines of our cars work in order to drive them
  – We do not have to know how a method works in order to invoke it; we need to know what it does and what it needs to be given in order to do it, but not how it does it
  – Therefore, we can write complex software without having to know how parts of it actually work; we just have to know what these parts do when we use them