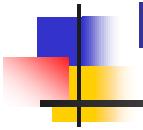
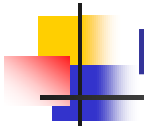


Introduction to Programming

7. Methods – Part 1 Static Methods (Subroutines)



Methods

- In this lecture we will start to look at Java methods
- We have seen that main() is a method and that it is the instructions in main() that are executed when we run a Java application
- We have also seen that in Java methods have to placed inside a class

What is a method?

- A method is a sequence of statements with a name
  - The statements are grouped in a block and can include declarations of local variables as well as executable statements such as if statements and while loops
  - The statements are executed whenever the method is *called*
  - The name of the method is used to call it

Instance and static methods

- Java has two kinds of methods
  - Static methods that belong to the class
    - main() is an example of a static method, as are the methods of TextIO such as put(), getln()
  - Instance methods that belong to an instance of the class (an object)
  - Will mainly look at static methods today
    - The textbook refers to these as *subroutines*, as they are rather like mini-programs

Calling a static method

- To call a static method, you give the name of the class it belongs to, followed by the name of the method and, in brackets, any parameters it expects (need the brackets even if no parameters)

```
TextIO.put("Enter a number: ");
int number = TextIO.getlnInt();
```


Calling an instance method

- To call an instance method, you give the name of the object it belongs to, followed by the name of the method and, in brackets, any parameters it expects (need the brackets even if no parameters)

```
String month = "November";
TextIO.put(month.toUpperCase());
```


Calling a method continued

- If you are calling a static method from within another static method of the same class, you do not need to name the class in the call
- Similarly, if you are calling an instance method of an object from another instance method of the same object you do not need to name the object

The value of subroutines

- A subroutine (in Java, a method) is a named set of instructions (executable statements)
- The subroutine name can be used in a program, in place of the instructions, whenever they have to be carried out
- In a well-designed program, the set of instructions in a subroutine are related and perform some conceptually single action (e.g. draw an object, calculate a value)
- Subroutines simplify programs by giving an action a name and allowing the programmer to execute the action without having to consider the details of how it is performed
- Subroutines can be called as many times as needed, so also simplify programs by re-using code

Subroutines as “Black Boxes”

- A subroutine
  - hides it contents from the client code that uses it
    - “black box” as you can’t see inside it
  - has a visible *interface* – the information you must have in order to use it
    - includes its name and what information needs to be passed in to the subroutine
    - also states what the subroutine does

Example subroutine interface

```
public class TextIO {
   ...
 /**
   * Write a single value to the current output
   * destination, using the default format
   * and no extra spaces.  This method will
   * handle any type of parameter, even one
   * whose type is one of the primitive types.
   */
   public static void put(Object x);
   ...
}
```


More on subroutine interfaces

- Subroutine interface must be clear and well-defined
  - forms a *contract* between the client (who calls the subroutine) and the implementer (who writes the subroutine)
    - tells client everything needed to use it
    - tells implementer everything needed to implement it

Method Signatures in Java

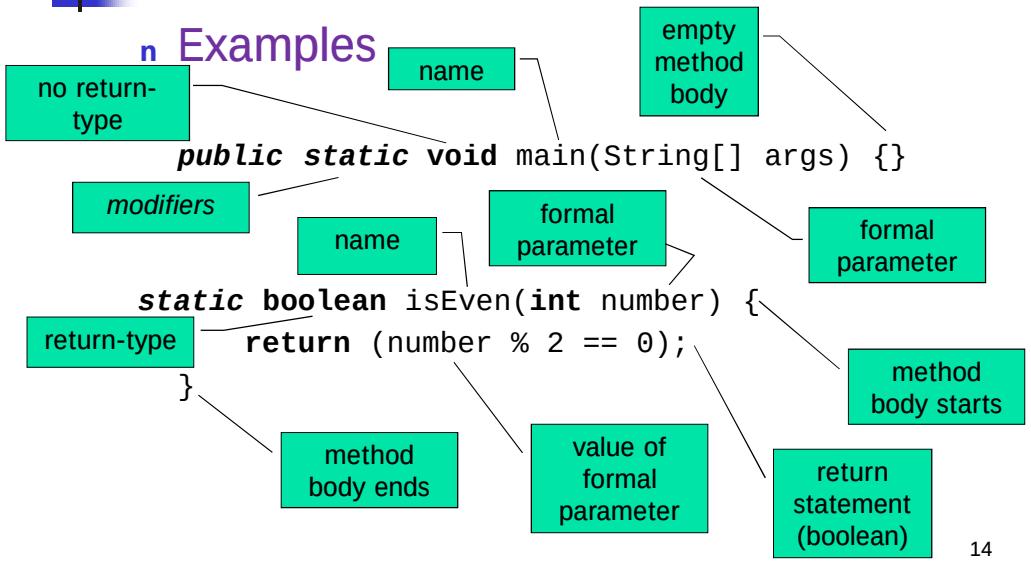
- The *signature* of a method comprises
  - the **name** of the method
    - this must be an identifier
  - the **types** of its *parameters* and the **order** in which they appear
    - Examples: putln(int) compare(String,String)
- No two methods in the same class can have the same signature
- Methods in the same class can have the same name if their parameters are different (overloading)

Method Declaration

- In addition to the signature information, the method declaration also includes
  - modifiers (e.g. **public**, **static)**
  - return type (the type of value it returns, or **void**)
  - parameter names (formal parameters)
    - a list of variable declarations (type, which is part of the signature, and name) separated by commas
    - parameter list must be between parentheses (brackets)
    - brackets must be there even if no parameters
  - method body (a block)

Method Declarations

- Examples



Method Declarations

- More Examples

```
public static void sayHello() {
    System.out.println("Hello");
}

int compare(int first, int second) {
    if (first < second) return -1;
    else if (first == second) return 0;
    else return 1;
}
```

no formal parameter

no modifiers

two formal parameters

compare() is an *instance method* as it is not declared as static

Method Execution

- The body of the method contains the set of instructions it names (this is the *implementation* of the method)
- As already noted, the body only executes when the method is *called*
- The method name is used to call it, with an argument supplied for each parameter in the method declaration
- Arguments are in parentheses (brackets)
  - Brackets must be included even if no parameters

Method body

- Is a block
- Can declare local variables, and must initialise them before using them
- Can contain almost any statements (**if**, **while**, **try-catch** *etc*.) but cannot include declarations of methods or of enums
- Can throw exceptions
- If a method returns a value (is not **void**) it must include at least one return statement
  - return statement terminates the method
  - must not be possible to reach end of method without executing a return statement (or throwing exception)

Method bodies

```
public class Evens {
   static boolean isEven(int number) {
       return (number % 2 == 0);
   }
   public static int getRandomEven() {
       int result;
       do {
             result = (int)(Math.random()*10000);
       } while (!isEven(result));
       return result;
   }
}
```

Question: What does this method do?

call static method from Math class

call static method from this class

Access Modifiers

- Control where the method is visible
  - **public** – visible whereever the class is visible (can be called from anywhere)
  - default – when no access modifier is used, also called *package-private*, visible only in the package that contains the class that the method is declared in (will look at packages later)
  - **private** – only visible inside the class that contains it

Access modifiers on methods

```
public class EvensVersion2 {
   private static boolean isEven(int number) {
         return (number % 2 == 0);
   }
   public static int getRandomEven() {
         int result;
         do {
            result =
               (int)(Math.random()*10000);
         } while (!isEven(result));
         return result;
   }
}
```

Hidden from all other classes

Visible to all other classes

call private helper method

Method parameters

- When method is called, the value of each argument (or *actual parameter*) is copied in to the corresponding formal parameter

```
public static int isEven(int number) {...}
public static void main(String[] args) {
    int x = 3; int y = 4;
    if (isEven(10) && isEven(x) && isEven(x+y))
{...}
}
```

actual parameter

actual parameter

actual parameter

Method parameters 2

- Value of the actual parameter is copied in to the formal parameter. Changes to the formal parameter do *not* affect the value of the actual.

```
public static void increment(int number) {
    number++;
}
public static void main(String[] args) {
    int x = 3;
    increment(x);
    System.out.println(x);  // prints 3, not 4
}
```

formal parameter

actual parameter

GuessingGame example (p142 of the book by Eck)

- A simple example of a Java application that uses a subroutine
  - Also illustrates the development of the algorithm using stepwise refinement
- The playGame() method contains all the logic to play one guessing game, the main() method contains a loop that calls playGame() repeatedly, for as long as the user wants to play

Pseudocode for playGame() 1

Pick a random number

while the game is not over:

  Get the user's guess

  Tell the user whether the guess is high, low, or correct.

Pseudocode for playGame() 2: adding detail

```
Let computersNumber be a random number between 1 and 100
Let guessCount = 0
while (true):
    Get the user's guess
    Count the guess by adding 1 to guess count
    if the user's guess equals computersNumber:
        Tell the user they won
        break out of the loop
    if the number of guesses is 6:
        Tell the user they lost
        break out of the loop
    if the user's guess is less than computersNumber:
        Tell the user the guess was low
    else if the user's guess is higher than computersNumber:
        Tell the user the guess was high
```


The playGame() method

- Declare and initialise the variables

```
static void playGame() {
   int computersNumber; // A random number picked by the
                        // computer.
   int usersGuess; // A number entered by user as a guess.
   int guessCount; // Number of guesses the user has made.
   computersNumber = (int)(100 * Math.random()) + 1;
   // The value assigned to computersNumber is a randomly
   // chosen integer between 1 and 100, inclusive.
   guessCount = 0;
```


playGame() continued 1

```
TextIO.putln();
TextIO.put("What is your first guess? ");
while (true) {
  usersGuess = TextIO.getInt(); // Get the user's guess.
  guessCount++;
  if (usersGuess == computersNumber) {
     TextIO.putln("You got it in " + guessCount
      + " guesses! My number was " + computersNumber);
     break; // The game is over; the user has won.
  }
  // continued on next slide, still in the loop
```


playGame() continued 2

```
  if (guessCount == 6) {
     TextIO.putln("You didn't get the number in 6 guesses.");
     TextIO.putln("You lose. My number was " + computersNumber);
     break; // The game is over; the user has lost.
  }
  // If we get to this point, the game continues.
  // Tell the user if the guess was too high or too low.
  if (usersGuess < computersNumber)
     TextIO.put("That's too low. Try again: ");
  else if (usersGuess > computersNumber)
     TextIO.put("That's too high. Try again: ");
  }
  TextIO.putln();
  } // end of playGame()
```

The main() method

```
public static void main(String[] args) {
   TextIO.putln("Let's play a game. I'll pick a number ");
   TextIO.putln("between 1 and 100, you try to guess it.");
   boolean playAgain;
   do {
      playGame(); // call subroutine to play one game
      TextIO.put("Would you like to play again? ");
      playAgain = TextIO.getlnBoolean();
   } while (playAgain);
   TextIO.putln("Thanks for playing. Goodbye.");
} // end of main()
```


Next week

- Will continue discussion of subroutines
- Reading: sections 4.3, 4.4 and 4.7 of the book

Questions on today's lecture?