Parameters and Objects

Chapter 3
A Class Metaphor

Classes, like blueprints, describe
An Object Metaphor

Objects, like houses, are usable
Attributes and Behaviors

Features or members

- Attributes
  - Characterize or distinguish an object
  - Are a data values (variables) held by objects
  - Each object maintains a private copy of each attribute value

- Behaviors
  - Are what an object can do (the services or operations provided by an object)
  - Are implemented as methods
  - Methods are called through or bound to an object, called “this object”
  - This object is the default target of the method (i.e., the data in this object is accessed by name only)
Access Control Modifiers

Control access to a class’s features

--- **private**: class scope; methods and instance variables

--- --- **default** (aka *friendly*): class and package scope; methods, instance variables, and classes

--- --- --- **protected**: class, package, and subclass scope; methods and instance variables

--- --- --- --- **public**: global scope; methods, instance variables, and classes
Methods

The active part of a class

```java
int a = 100;
int b = 25;
```

- When used or called, a method can be treated as a black box
  - User cares only about what the method does, not about how (performance and errors aside)
  - The method's interface or signature is a description of how to connect with or call the method

- Method input (parameters) may be
  - Constants
  - Variables
  - Expressions
Method Execution

Call and return sequences

main()
{
method1()
{
method1()
{
method2();
method3();
}
return;
}
method2();
method3();
return;
}

return;
Method Definition Syntax

AKA declaration syntax

- Always defined inside a class but can **not** be nested

- **Header**
  - Modifiers 0 or more; common: public, private, etc.
  - Return value type required, may be void
  - Name Java style is “Camel notation”
  - Argument list *formal parameters* – ‘(’ and ‘)’ are **always** required
  - Definitions require typing info; documentation shown in this format

- **Body**
  - Between ‘{’ and ‘}’
  - Contains any number of legal Java statements (including 0)

```java
public class Employee
{
  public Manager getBoss(int depart, int when) { ...... }
  public static int getPhoneNumber() { ...... }
}
```
Method Call Syntax

AKA invocation, dispatch

- Does **not** include modifiers or typing information
- Actual parameters may be expressions (constants, variables, method calls, or arithmetic expressions using any of the 3)
- Non-static methods are called through an object
- Static methods are called through the class

```java
Employee worker = new Employee( );
Manager boss = worker.getBoss(416, 2009 + 47); // non-static
int number = Employee.getPhoneNumber( ); // static
```
Method Return Values

Getting a value from a method

- Methods terminate by “falling off the end” or calling `return`
  - `return` may be called anywhere in the method
  - Method return values are specified with the `return` statement
    (i.e., using return is the only way to get a value returned from a
    method to the calling scope)

- A method that does not return a value has a “return” type of `void`

- Return values may be ignored
Method Overloading

Methods with the same name

- Overloaded methods
  - Defined in the same class
  - Have the same name
  - Must have different signatures
  - May have different return types but cannot overload on return type
  - Should perform similar tasks
  - println(char)
  - println(int)
  - println(double)
  - println(String)

```
type name(parameters) {
    local variables;
    statements;
}
```
Constructors

A special-purpose method

```java
public class Point {
    int x, y;

    public Point() {
        x = 100;
        y = 200;
    }

    public Point(int px, int py) {
        x = px;
        y = py;
    }
}
```

- Have the same name as the class
- No return type or value
- May be complex or may be simple
- May be overloaded

*Default constructors* do not have parameters

```java
Point p1 = new Point();
Point p2 = new Point(50, 25);
```
Value Vs. Reference

Accessing data

- Primitive data are accessed by value
  - \(x + y\)
  - value stored in \(x\) + value stored in \(y\)

- Objects are accessed by reference
  - A reference is a memory address
  - “null” is a special (empty) address
  - \(\text{Circle } c;\) // define reference var
  - \(c = \text{new Circle}(10,0,0);\) // instantiation only
  - \(\text{Circle } c = \text{new Circle}(10,0,0);\) // both in 1 statement

```java
int x = 10;
int y = 20;
x = 10
y = 20
```
The *this* Reference

Object self reference

- “Secret” argument passed to each non-static method
- Points to the calling object; binds it’s data to the method
- Used for self reference

```java
class Epi
{
    double r, a;
    void draw()
    {
        x = f1(r,a);
        y = f2(r,a);
    }
}

class Spiro
{
    Epi e1 = new Epi(100, 17);
    Epi e2 = new Epi(37, 19);
    Epi e3 = new Epi(47, 39);
    void display()
    {
        e2.draw();
    }
}
```
Packages

Organizing libraries

- Java organizes related classes into a “package”
- A Java file can state to which package it belongs with the `package` keyword
- Each package occupies a separate subdirectory
- Package names must be the same as the name of the directory in which the compiled (i.e., bytecode) files reside
- Each compiled class occupies a separate file in the package subdirectory
- Programs access the contents of a package by importing
  - `import packagename.classname;`
  - `import packagename.*;`
Scope

Where an identifier (symbol) is visible

```java
public void method( )
{
    int n;  // local variable
    ...
    {
        int k;
    }  // k goes out of scope
}  // n goes out of scope

public void method( )
{
    int n;  // local variable
    ...
    {
        int k;
        int n;  // compiler error
    }
}
```

// a common scoping error -
// local variable “manager” shadows
// instance variable “manager”

```java
public class MyClass
{
    private Person manager;

    public MyClass(String name)
    {
        Person manager = new Person(name);
    }
}
```
Creating object aliases

- Object names are references that “point” to an object
- Multiple names (aliases) may reference the same object
  - Created by assignment
  - Created by method calls

```java
object name          object type (class name)
Circle c = new Circle(100,20,10);
Circle t = c;       // doesn’t duplicate
draw(c);            // method call

public void draw(Circle s)
{
    ...
}
```
Pass By Value: Simple Types

Illustrated with int

```c
int x = 100;
inc(x);

void inc(int a)
{
    a += 100;
}
```

- Define `x` as 100.
- Call `inc(x)`.
- Inside `inc()`, `a` is incremented by 100.
- Return `x`.

**Time:**
```plaintext
define x | call inc | run inc | return
100    | 100    | 200    | 100
```
Pass By Value: Objects

Assume class Employee

- Java only has pass by value
- The value passed is an address

Employee x = new Employee("Fred");

call update

update(x);

void update(Emp a)
{
    a = new Employee("Sid");
}

define x

return

time
Pass By Value: Updating Objects

Class Employee has an instance variable name

Employee x =
    new Employee("Fred");

update(x);

void update(Emp a) {
    a.name = "Sid";
}

define x  call update  run update  return

time
Strings Revisited

Two views

- Can be treated as an atomic entity (chap 2, slide 8)
- Can be treated as a list or string of characters
  - Individual characters accessed by position (i.e., index or subscript)
  - Legal index values range from 0 to length - 1
  - char charAt(int index)
  - char[ ] toCharArray()
  - String substring(int beginIndex)
  - String substring(int beginIndex, int endIndex)
  - Throws an IndexOutOfBoundsException if index < 0 or index >= length

String s = "Hello";
Exceptions

Dealing with runtime errors

- Claiming an exception
  - `throws` Introduces a comma-separated list of exceptions that a method might throw

- Throwing an exception
  - `throw` Launch an exception object to indicate that an exception situation has occurred

- Catching an exception
  - `try` Define a block of code that anticipates exceptions
  - `catch` Define a block of code that deals with a specific exception
  - `finally` Define a block of code that executes regardless of whether an exception is generated or not
Input with the Scanner Class

Details and examples in java.util.Scanner

- **Some Constructors**
  - Scanner(InputStream source)
  - Scanner(String source)

- **Some Methods**
  - double nextDouble()
  - int nextInt()
  - String nextLine()
  - boolean hasNextDouble()
  - boolean hasNextInt()
  - boolean hasNextLine()

```java
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
while (sc.hasNextLine())
{
    String line = sc.nextLine();
    // process line
}
```
Input with showInputDialog

Pause for input; returns a String

- String `showInputDialog`(Component parent, Object message, String title, int messageType)

String input = JOptionPane.showInputDialog(null,
   "Please enter a number of pennies",
   "Money Input", JOptionPane.QUESTION_MESSAGE);

int pennies = Integer.parseInt(input);