This chapter discusses

- Implementing class definitions.
- How to store data in an object and how to write method bodies.
- Basics of *statements* and *expressions*.
Component Variables

- Objects store information in variables.

- variable: a portion of memory reserved for storing a value.

- component variable: a variable that is a permanent part of an object; memory space for the variable is allocated when the object is created, and the variable exists as long as the object exists. (often called the handle or pointer to the object)
Component Variables (cont.)

- Syntax:
  ```java
  private variableType variableName;
  ```

- Information should not be directly available to clients; clients should access information only through queries and commands.
Back to the *Counter* class

- Remember that the class needed to ‘Know’ its properties:
  - the value of count.
- We will call this variable `tally`.

```java
public class Counter {
    ...
    private int tally;
    ...
} // end of class Counter
```

- Every *Counter* object will have a component that is an `int` variable named `tally`. 
public class Explorer {
    ...
    private String playerName;
    private rooms.Room room;
    private int strengthPoints;
    private int staminaPoints;
    ...
} // end of class Explorer
### Explorer

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>playerName</td>
<td>&quot;Marjorie&quot;</td>
</tr>
<tr>
<td>room</td>
<td></td>
</tr>
<tr>
<td>strengthPoints</td>
<td>10</td>
</tr>
<tr>
<td>staminaPoints</td>
<td>100</td>
</tr>
</tbody>
</table>

- **Room**
Constructors

- Every constructor should ensure that all of a newly created object’s component variables are initialized appropriately.
Named constants

- Sometimes data types provided by Java do not adequately represent something you wish to model.
- Example: Card suit--spade, heart, diamond, club.
- Solution: Represent a suit with integers.

spade=4 heart=3 diamond=2 club=1
Named constants (cont.)

- You can attach names to these values, and then refer to them by name.

- Example:

  ```java
  public final static int SPADE = 4;
  public final static int HEART = 3;
  public final static int DIAMOND = 2;
  public final static int CLUB = 1;
  ```

- Syntax: `public final static type identifier = constantExpression`
Named constants (cont.)

- Now you can refer to the values by their names.
  
  ```java
  if (discard.suit() == Card.SPADE)...
  ```

- **named constant**: a value that has an identifier (name) associated with it; the associated identifier can be used to refer to the value.

- By convention, we use upper-case identifiers for named constants.
Implementing functionality

- **statement**: a language construct that describes an action for the processor to perform.
- The method body consists of a sequence of one or more statements.
Implementing functionality (cont.)

- Queries perform one action which is to give the value of a component variable.
- Example:

  ```java
  public int getCount () {
      return tally;
  }
  ```

- Every query ends with the execution of a return statement.
- Syntax: `return expression`
Implementing functionality (cont.)

**Explorer Queries:**

```java
public String name () {
    return playerName;
}

public rooms.Room getLocation () {
    return room;
}

public int getStrength () {
    return strengthPoints;
}

public int getStamina () {
    return staminaPoints;
}
```
Arithmetic expressions

- **expression**: a language construct that describes how to compute a particular value.
- Expressions that evaluate to type `byte`, `short`, `int`, `long`, `float`, and `double`, are collectively called **arithmetic expressions**.
Operators

- **Unary operators**: involves manipulation of one value.
- Affixing ‘+’ or ‘-’ can be considered a unary operator. Affixing the ‘+’ does nothing. Affixing the ‘-’ negates the value.
- **Example**: If \( a = 5 \) and \( b = -4 \)
  
  \[ +a = 5, \quad -a = -5, \quad +b = -4, \quad -b = 4 \]
Operators (cont.)

- **binary operators**: combine 2 expressions (operands).
- Operators include `'+', '-', '*', '/', '%'.
- The not-so-obvious operators:
  - `'*'` - multiplication
  - `'/ '` - division
  - `%` - modular division (remainder).
The “/” denotes *division* when applied to two floating point operands, but *integer quotient* when applied to two integer operands.

\[
\begin{align*}
2.0/4.0 & \rightarrow 0.5 \\
2/4 & \rightarrow 0 \\
5.0/4.0 & \rightarrow 1.25 \\
5/4 & \rightarrow 1
\end{align*}
\]
Operator Precedence

- If \( i1 = 10 \), what is the order of evaluation in \( i1 + 10 \times 2 \)?
- Unary + and – have higher precedence than the binary operators.
- \( \ast, \div, \% \) have higher precedence than the binary operators +, –.
- If two operators have equal precedence, operations are performed left to right. i.e. \( 10 \div 5 \times 3 = 6 \)
- Parentheses are used to override precedence. i.e. \( 10 \div (5 \times 3) \)
Casting

- Occasionally we must convert a value to a different type to perform certain operations.
- **Syntax:** (type) expression
- **Example** (double) age
Casting (cont.)

- Consider these expressions:
  \[
  10/40 = 0 \\
  (\text{double})10/(\text{double})40 = 0.25 \\
  \\
  10.0/40.0 = 0.25 \\
  (\text{int})10.0/(\text{int})40.0 = 0
  \]

- Cast operators have *higher precedence* than arithmetic operators.
Assignment statement

- **assignment**: a statement that instructs the processor to compute a value and to store it in a variable.
- It is denoted by ‘='. Note: The ‘=' sign does not mean mathematical equality.
- **Syntax**: `variableName = expression`

```java
public void reset() {
    tally = 0;
}

public void stepCount () {
    tally = tally + 1;
}
```
Implementing the Constructor

- The constructor is invoked when creating a new instance of an object.
- It initializes the component variables of the object.

```java
public counter () {
    tally = 0;
}
```
package counters;

/**
 * A simple integer counter.
 */
public class Counter {

    // Constructors:

    /**
     * Create a new Counter.
     */
    public Counter () {
        tally = 0;
    }

    // Queries:

    /**
     * Current count; the number of items counted.
     */
    public int count () {
        return tally;
    }

    // Commands:

    /**
     * Increment the count by 1.
     */
    public void stepCount () {
        tally = tally + 1;
    }

    /**
     * Reset the count to 0.
     */
    public void reset () {
        tally = 0;
    }
}
// Private components:

private int tally; // current count

} // end of class Counter
Using parameters

- Parameters that are passed in are used in making some calculations.

```java
public void takeHit (int hitStrength){
    staminaPoints = staminaPoints - hitStrength;
}
```

- **Parameter variable**: a variable that is created when a method is invoked, and destroyed when the processor finishes executing the method.

- **Example**: hitStrength
Explorer

- playerName: "Marjorie"
- room
- strengthPoints: 10
- staminaPoints: 100

Room

instance variables of the object

- hitStrength: 5

automatic variable

argument value supplied by the client
**Explorer class**

```java
double void setName(String newName)
{
    playerName = newName;
}

double void setLocation(rooms.Room newRoom) {
    location = newRoom;
}
```
The *Explorer* constructor

```java
public Explorer (String name,
                 rooms.Room location,
                 int hitStrength,
                 int stamina) {
    playerName = name;
    room = location;
    strengthPoints = hitStrength;
    staminaPoints = stamina;
}
```
Invoking a method: acting as a client

- Consider the command `strike`.

```java
public void strike (denizens.Denizen monster) {...}
```
We now want to change the state of the monster’s stamina. We use the `takehit` command to do this.

Recall how to invoke a method:

```java
object.command (arguments)
```

Therefore:

```java
public void strike (denizens.Denizen monster) {
    monster.takeHit (strengthPoints);
}
```
The `strike` command is both a server and a client. It is the server for whatever object invokes the `strike` command; it is the client when invoking the `takeHit` command.
package explorer;

/**
 * A maze game player.
 */
public class Explorer {

    // Constructors:

    /**
     * Create a new Explorer.
     */
    public Explorer (String name, rooms.Room location,
            int hitStrength, int stamina) {
        playerName = name;
        room = location;
        strengthPoints = hitStrength;
        staminaPoints = stamina;
    }

    // Queries:

    /**
     * Name of this Explorer.
     */
    public String name () {
        return playerName;
    }

    /**
     * Room in which this Explorer is currently located.
     */
    public rooms.Room location () {
        return room;
    }

    /**
     * Damage (hit points) this Explorer does when
     * striking.
     */
    public int strength () {
        return strengthPoints;
    }
}
continued
return strengthPoints;
}

/**
 * Damage (hit points) required to defeat this
 * Explorer.
 */
public int stamina () {
    return staminaPoints;
}

// Commands:

/**
 * Change the name of this Explorer to the specified
 * String.
 */
public void changeName (String newName) {
    playerName = newName;
}

/**
 * Move to the specified Room.
 */
public void move (rooms.Room newRoom) {
    room = newRoom;
}

/**
 * Receive a blow of the specified number of hit
 * points.
 */
public void takeHit (int hitStrength) {
    staminaPoints = staminaPoints - hitStrength;
}

/**
 * Strike the specified Denizen.
 */
public void strike (denizens.Denizen monster) {
    monster.takeHit(strengthPoints);
}
// Private components:

private String playerName;  // name
private rooms.Room room;   // current location
private int strengthPoints; // current strength
                            // (hit points)
private int staminaPoints;  // current stamina
                            // (hit points)

} // end of class Explorer
More on methods

- Arguments provided in a method are in general *expressions*.
- A method invocation must provide an argument of the appropriate type for each parameter. i.e.

```java
public void move(int direction,
                 double distance)
```

requires an int and double

```java
object.move(90, 2.5);
```
More on methods (cont.)

- A command invocation is a form of a *statement*. A query is a form of *expression*, since it produces a value.

- Examples:
  
  ```java
  i = c.getCount();
  i = c.getCount() + 10;
  ```
Local variables

- Local variables are automatic variables created as part of a method execution, used to hold intermediate results needed while the method is active.

- They are not initialized by the client. They must be initialized in the method.

- **Syntax:**
  ```
  type identifier;
  type identifier = expression;
  ```
CashRegister class

```java
public double netPrice
    (double grossPrice,
     double taxRate,
     double discountRate)
{
    double tax;
    double discount;
    tax = grossPrice * taxRate;
    discount = grossPrice * discountRate;
    return grossPrice + tax - discount;
}

- tax and discount are local automatic variables.
```
**Local Variable:**
- Defined inside a method.
- Exists only while the method is being executed.
- Can be accessed only from the method.
- Is only meaningful during execution of the method.
- Contains some intermediate value needed only during execution of the method; its value is not part of the object’s state.

**Instance Variable:**
- Defined outside any method.
- Exists as long as the object exists.
- Can be accessed from any method in the class.
- Has a meaningful value at any time during the life of the object, whether the object is actively doing something or not.
- Represents a property of the object; its value is part of the object’s state.
We’ve covered

- How to write a simple class implementation.
- Component variables, named constants, automatic variables, and local variables.
- Method bodies.
- The `return` statement and the assignment (=) statement.
- Command and query invocation.

```plaintext
object.commandOrQuery(arguments)
```
Glossary

**arithmetic expression:** an expression that produces an integer or floating point value when evaluated.

**assignment statement:** a statement that instructs the processor to compute a value and store it in a variable.

**associativity rule:** a rule that specifies the order in which un-parenthesized operators of equal precedence are applied in the evaluation of an expression.

**automatic variable:** a variable that is created when a method is invoked, and deallocated when the method is completed.

**binary operator:** an operator that requires two operands; that is, computes a value from two given values. Also called a *dyadic* operator.

**cast:** an operation in which a value of one type is converted into a value of a different type.

**component:** an instance variable.

**expression:** a language construct that specifies how to compute a particular value.

**instance variable:** a variable that contains data stored as part of an object’s state. An instance variable is allocated when the object is created, and exists as long as the object does.
local variable: an automatic variable created as part of a method execution, used to hold intermediate results needed during the computation.

method body: the sequence of statements that comprise the implementation of a method. When the method is invoked, the processor executes the statements that make up the method body.

named constant: a value that has an identifier (a name) associated with it; the associated identifier can be used to refer to the value.

operand: a value given to an operator, and used in the computation of another value.

operator precedence rules: rules that specify order in which operators are applied in the evaluation of an expression.

return statement: a statement that specified the value to be delivered to the client. A return statement is the last statement executed in a query method.

statement: a language construct that describes an action for the processor to perform.