Objectives

This week we will study how to create your own classes including:

- Local and instance variables
- Value-returning methods
- Private and public methods, data members
- Passing primitive data and objects to a method
Why Programmer-Defined Classes

- Using just the String, GregorianCalendar, JFrame and other standard classes will not meet all of our needs. We need to be able to define our own classes customized for our applications.
- Learning how to define our own classes is the first step toward mastering the skills necessary in building large programs.
- Classes we define ourselves are called programmer-defined classes.
Using the Bicycle Class

class BicycleRegistration {
    public static void main(String[] args) {
        Bicycle bike1, bike2;
        String owner1, owner2;

        bike1 = new Bicycle(); //Create and assign values to bike1
        bike1.setOwnerName("Lance Armstrong");

        bike2 = new Bicycle(); //Create and assign values to bike2
        bike2.setOwnerName("Ada Augusta");

        owner1 = bike1.getOwnerName(); //Output the information
        owner2 = bike2.getOwnerName();

        System.out.println(owner1 + " owns a bicycle.");
        System.out.println(owner2 + " also owns a bicycle.");
    }
}
The Bicycle Class

class Bicycle {

    // Data Member
    private String ownerName;

    // Constructor: Initializes the data member
    public Bicycle() {
        ownerName = "Unknown";
    }

    // Returns the name of this bicycle's owner
    public String getOwnerName() {
        return ownerName;
    }

    // Assigns the name of this bicycle's owner
    public void setOwnerName(String name) {
        ownerName = name;
    }
}
Multiple Instances

- Once the Bicycle class is defined, we can create multiple instances.

Sample Code
Multiple Instances

Once the Bicycle class is defined, we can create multiple instances.

```java
Bicycle bike1, bike2;
```

Sample Code
Multiple Instances

Once the Bicycle class is defined, we can create multiple instances.

```java
Bicycle bike1, bike2;
bike1 = new Bicycle();
```
Multiple Instances

- Once the Bicycle class is defined, we can create multiple instances.

```java
Bicycle bike1, bike2;
bike1 = new Bicycle();
```

Sample Code
Once the Bicycle class is defined, we can create multiple instances.

```java
Bicycle bike1, bike2;
bike1 = new Bicycle();
bike1.setOwnerName("Lance Armstrong");
```

Sample Code
Once the Bicycle class is defined, we can create multiple instances.

Sample Code

```java
Bicycle bike1, bike2;
bike1 = new Bicycle();
bike1.setOwnerName("Lance Armstrong");
bike2 = new Bicycle();
```
Once the Bicycle class is defined, we can create multiple instances.

```java
Bicycle bike1, bike2;
bike1 = new Bicycle();
bike1.setOwnerName("Lance Armstrong");
bike2 = new Bicycle();
```

**Sample Code**
Once the `Bicycle` class is defined, we can create multiple instances.

```java
Bicycle bike1, bike2;
bike1 = new Bicycle();
bike1.setOwnerName("Lance Armstrong");
bike2 = new Bicycle();
bike2.setOwnerName("Ada Augusta");
```

Sample Code
Program Structure and Source Files

To run the program:
1. `javac Bicycle.java` (compile)
2. `javac BicycleRegistration.java` (compile)
3. `java BicycleRegistration` (run)

There are two source files. Each class definition is stored in a separate file.
Class Diagram for Bicycle

Method Listing
We list the name and the data type of an argument passed to the method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bicycle()</td>
<td></td>
</tr>
<tr>
<td>getOwnerName()</td>
<td></td>
</tr>
<tr>
<td>setOwnerName(String)</td>
<td></td>
</tr>
</tbody>
</table>
Template for Class Definition
Template for Class Definition

class

{ import statements

}


Template for Class Definition

class {

import statements

}
Template for Class Definition

class
{

}

Import Statements
Class Comment
Class Name
Template for Class Definition

class 

{ 

}
Data Member Declaration

```plaintext
<modifiers> <data type> <name> ;
```
Data Member Declaration

```java
private String ownerName;  
```
Data Member Declaration

```java
<modifiers> <data type> <name> ;
```

```
private String ownerName;
```
Data Member Declaration

```
<modifiers>  <data type>  <name>  ;
```

```
private    String    ownerName    ;
```

Note: There's only one modifier in this example. This is visibility modifier.
Method Declaration

<modifier>  <return type>  <method name>  (  <parameters>  ){
  <statements>
}

Method Declaration

```java
public void setOwnerName ( String name ) {
    ownerName = name;
}
```
Method Declaration

```java
public void setOwnerName ( String name ) {
    ownerName = name;
}
```
Execution flow

- Our programs begin execution at the first statement in the main method.
- Statement are executed in order.
- When a method is called,
  - the execution moves to the first line of that method
  - each statement is executed in order, until
  - we get a return statement or the end of the method
  - then control returns back to the caller.
Example flow

class Test {
    public static void main (String[] args) {
        println("inside main");
        methodA();
        println("back from methodA");
    }
}

public void methodA () {
    println("inside methodA");
    methodB();
    println("back from methodB");
}

public void methodB () {
    println("inside methodB");
}
class Test {
    public static void main (String[] args) {
        println("inside main");
        methodA();
        println("back from methodA");
    }
}

public void methodA () {
    println("inside methodA");
    methodB();
    println("back from methodB");
}

public void methodB () {
    println("inside methodB");
}
Example flow

class Test {
    public static void main (String[] args) {
        println(“inside main”);
        methodA();
        println(“back from methodA”);
    }
}

public void methodA () {
    println(“inside methodA”);
    methodB();
    println(“back from methodB”);
}

public void methodB () {
    println(“inside methodB”);
}

```java
class Test {
    public static void main (String[] args) {
        println("inside main");
        methodA();
        println("back from methodA");
    }
}
```

```java
public void methodA () {
    println("inside methodA");
    methodB();
    println("back from methodB");
}
```

```java
public void methodB () {
    println("inside methodB");
}
```
class Test {
    public static void main (String[] args) {
        println("inside main");
        methodA();
        println("back from methodA");
    }
}

public void methodA () {
    println("inside methodA");
    methodB();
    println("back from methodB");
}

public void methodB () {
    println("inside methodB");
}
class Test {
    public static void main (String[] args) {
        println("inside main");
        methodA();
        println("back from methodA");
    }
}

public void methodA () {
    println("inside methodA");
    methodB();
    println("back from methodB");
}

public void methodB () {
    println("inside methodB");
}
Example flow

class Test {
    public static void main (String[] args) {
        println("inside main");
        methodA();
        println("back from methodA");
    }
}

class Test {
    public void methodA () {
        println("inside methodA");
        methodB();
        println("back from methodB");
    }
}

class Test {
    public void methodB () {
        println("inside methodB");
    }
}
```java
class Test {
    public static void main(String[] args) {
        System.out.println("inside main");
        methodA();
        System.out.println("back from methodA");
    }
}

public void methodA() {
    System.out.println("inside methodA");
    methodB();
    System.out.println("back from methodB");
}

public void methodB() {
    System.out.println("inside methodB");
}
```
Example flow

```java
class Test {
    public static void main (String[] args) {
        println("inside main");
        methodA();
        println("back from methodA");
    }
}

public void methodA () {
    println("inside methodA");
    methodB();
    println("back from methodB");
}

public void methodB () {
    println("inside methodB");
}
```
Example flow

class Test {
    public static void main (String[] args) {
        println("inside main");
        methodA();
        println("back from methodA");
    }
}

public void methodA () {
    println("inside methodA");
    methodB();
    println("back from methodB");
}

public void methodB () {
    println("inside methodB");
}
Example flow

class Test {
    public static void main (String[] args) {
        println("inside main");
        methodA();
        println("back from methodA");
    }
}

public void methodA () {
    println("inside methodA");
    methodB();
    println("back from methodB");
}

public void methodB () {
    println("inside methodB");
}
public void methodA () {
    println("inside methodA");
    methodB();
    println("back from methodB");
}

public void methodB () {
    println("inside methodB");
}

class Test {
    public static void main (String[] args) {
        println("inside main");
        methodA();
        println("back from methodA");
    }
}
class Test {
    public static void main (String[] args) {
        println("inside main");
        methodA();
        println("back from methodA");
    }
}

public void methodA () {
    println("inside methodA");
    methodB();
    println("back from methodB");
}

public void methodB () {
    println("inside methodB");
}
Return values

- Most methods return a value of a given data type. These are called value returning methods.
- If no value is returned, then the return type is declared to be `void`.
- Such a method is called a `void method`.
- Methods return values by using the statement:
  ```java
  return <expression>;
  return bike1;
  ```
- The type of the expression must be compatible with the return type.
- A `return` causes the method to return to the caller.
Types of methods
Types of methods

- Constructor
  - These are called in order to create an instance of the class.
  - There must always be a constructor method. If none is defined, then the compiler automatically adds one!
  - Default constructor has no parameters.
  - A constructor does not have a return type (or return statement).
Types of methods

- **Constructor**
  - These are called in order to create an instance of the class.
  - There must always be a constructor method. If none is defined, then the compiler automatically adds one!
  - Default constructor has no parameters.
  - A constructor does not have a return type (or return statement).

- **Accessor methods**
  - These are used to retrieve values of variables of an object.
Types of methods

- **Constructor**
  - These are called in order to create an instance of the class.
  - There must always be a constructor method. If none is defined, then the compiler automatically adds one!
  - Default constructor has no parameters.
  - A constructor does not have a return type (or return statement).

- **Accessor methods**
  - These are used to retrieve values of variables of an object.

- **Mutator methods**
  - These are used to modify (mutate) the values of variables of an object.
Types of methods

- **Constructor**
  - These are called in order to create an instance of the class.
  - There must always be a constructor method. If none is defined, then the compiler automatically adds one!
  - Default constructor has no parameters.
  - A constructor does not have a return type (or return statement).

- **Accessor methods**
  - These are used to retrieve values of variables of an object.

- **Mutator methods**
  - These are used to modify (mutate) the values of variables of an object.

- **Private methods**
  - These are only used by other methods of the class.
A constructor is a special method that is executed when a new instance of the class is created.

```java
public <class name> ( <parameters> ){
    <statements>
}
```
A constructor is a special method that is executed when a new instance of the class is created.

```java
public <class name> ( <parameters> ){
    <statements>
}
```

```java
public Bicycle ( ) {
    ownerName = "Unassigned";
}
```
A constructor is a special method that is executed when a new instance of the class is created.

```java
public <class name> ( <parameters> ){
    <statements>
}
```

```java
public Bicycle ( ) {
    ownerName = "Unassigned";
}
```
A constructor is a special method that is executed when a new instance of the class is created.

```java
public <class name> ( <parameters> ){
    <statements>
}
```

Example:

```java
public Bicycle () {
    ownerName = "Unassigned";
}
```
Example: Using Bicycle and Account

class SecondMain {

    //This sample program uses both the Bicycle and Account classes

    public static void main(String[] args) {

        Bicycle bike;
        Account acct;

        String myName = "Jane Java";

        bike = new Bicycle();
        bike.setOwnerName(myName);

        acct = new Account();
        acct.setOwnerName(myName);
        acct.setInitialBalance(250.00);

        acct.add(25.00);
        acct.deduct(50);

        //Output some information
        System.out.println(bike.getOwnerName() + " owns a bicycle and");
        System.out.println("has $ " + acct.getCurrentBalance() + " left in the bank");
    }
}
The Account Class

class Account {
    private String ownerName;
    private double balance;

    public Account() {
        ownerName = "Unassigned";
        balance = 0.0;
    }

    public void add(double amt) {
        balance = balance + amt;
    }

    public void deduct(double amt) {
        balance = balance - amt;
    }

    public double getCurrentBalance() {
        return balance;
    }

    public String getOwnerName() {
        return ownerName;
    }

    public void setInitialBalance(double bal) {
        balance = bal;
    }

    public void setOwnerName(String name) {
        ownerName = name;
    }
}
Multiple Constructors

- Suppose we don’t want to reset balances except during account creation.
- Can achieve this by defining another constructor:

```java
public Account(double initialBalance) {
    ownerName = "Unassigned";
    balance = initialBalance;
}
```

- New we can create and initialize once.
- We can have both defined at the same time.
Multiple constructors (contd.)

- Multiple constructors can exist at the same time.
- Which one gets executed?
- Depends upon the signature (arguments):
  ```java
  acct = new Account ();
  acct = new Account ("John Smith");
  ```
- This is called **overloading** (more later).
- Note: if you define one constructor, no default constructor is created.
Program Structure for SecondMain

To run the program:
1. javac Bicycle.java  (compile)
2. javac Account.java  (compile)
2. javac SecondMain.java  (compile)
3. java SecondMain  (run)

Note: You only need to compile the class once. Recompile only when you made changes in the code.
Calling a method

```java
class Sample {
    public static void main(String[] arg) {
        Account acct = new Account();
        ...
        acct.add(400);
        ...
    }
    ...
}

class Account {
    ...
    public void add(double amt) {
        balance = balance + amt;
    }
    ...
}
```
When a method is called, two things happen
- New storage is created for the parameters, and values are assigned to them.
- The program begins execution at the beginning of the method.
When a method is called, two things happen

- New storage is created for the parameters, and values are assigned to them.
- The program begins execution at the beginning of the method.

When a method ends (returns) execution resumes after the calling statement.
class Account {
    public void add(double amt) {
        balance = balance + amt;
    }
}

class Sample {
    public static void main(String[] arg) {
        Account acct = new Account();
        acct.add(400);
    }
}

Arguments and Parameters
Arguments and Parameters

- An argument is a value we pass to a method

```java
class Account {
    public void add(double amt) {
        balance = balance + amt;
    }
}

class Sample {
    public static void main(String[] arg) {
        Account acct = new Account();
        . . .
        acct.add(400);
        . . .
    }
    . . .
}
```
An argument is a value we pass to a method.

- A parameter is a placeholder in the called method to hold the value of the passed argument.

Arguments are also called actual parameters (vs. formal parameters in the method).
Matching Arguments and Parameters

```java
class Demo {
    public void compute(int i, int j, double x) {
        ... 
    }
}
```

```java
Demo demo = new Demo();
int i = 5; int k = 14;
demo.compute(i, k, 20);
```
Matching Arguments and Parameters

- The number of arguments and the parameters must be the same.

```java
Demo demo = new Demo();
int i = 5; int k = 14;
demo.compute(i, k, 20);
demo.compute(i, k, 20);
```

Passing Side

```java
class Demo {
    public void compute(int i, int j, double x) {
        ...
    }
}
```

Receiving Side
Matching Arguments and Parameters

- The number of arguments and the parameters must be the same.

- Arguments and parameters are paired left to right.

```java
class Demo {
    public void compute(int i, int j, double x) {
        // ... 
    }
}
```

```java
Demo demo = new Demo();
int i = 5; int k = 14;
demo.compute(i, k, 20);
```

3 arguments

Passing Side

3 parameters

Receiving Side
Matching Arguments and Parameters

- The number of arguments and the parameters must be the same
- Arguments and parameters are paired left to right

```java
class Demo {
    public void compute(int i, int j, double x) {
        ...
    }
}

Demo demo = new Demo();
int i = 5; int k = 14;
demo.compute(i, k, 20);
```
Matching Arguments and Parameters

- The number of arguments and the parameters must be the same
  - Arguments and parameters are paired left to right
  - The matched pair must be assignment-compatible (e.g. you cannot pass a double argument to an int parameter)

```java
class Demo {
    public void compute(int i, int j, double x) {
        // ... 
    }
}
```

```java
Demo demo = new Demo();
int i = 5; int k = 14;
demo.compute(i, k, 20);
```

3 arguments

3 parameters
Memory Allocation

Literal constant has no name

Passing Side

i 5
k 14
20
Memory Allocation

- Separate memory space is allocated for the receiving method.
Memory Allocation

- Separate memory space is allocated for the receiving method.
  - Values of arguments are passed into memory allocated for parameters.

**Passing Side**

- i: 5
- k: 14
- 20

**Receiving Side**

- i
- j
- x

Literal constant has no name
Memory Allocation

- Separate memory space is allocated for the receiving method.
  - Values of arguments are passed into memory allocated for parameters.
Passing Objects to a Method
Passing Objects to a Method

- As we can pass `int` and `double` values, we can also pass an object to a method.
Passing Objects to a Method

- As we can pass `int` and `double` values, we can also pass an object to a method.
- When we pass an object, we are actually passing the reference (name) of an object.
  - it means a duplicate of an object is NOT created in the called method.
class LibraryCard {
    private Student owner;
    public void setOwner(Student st) {
        owner = st;
    }
}

LibraryCard card2;
card2 = new LibraryCard();
card2.setOwner(student);

Passing Side

Receiving Side

State of Memory
Passing a Student Object

```java
class LibraryCard {
    private Student owner;
    public void setOwner(Student st) {
        owner = st;
    }
}
```

```java
LibraryCard card2;
card2 = new LibraryCard();
card2.setOwner(student);
```

Passing Side

Receiving Side

1 Argument is passed

State of Memory

St: 
- name: "Jon Java"
- email: "jj@javauniv.edu"

Card2: 
- owner: Student
- borrowCnt: 0

Class LibraryCard: 
- private Student owner
- public void setOwner(Student st) {
  owner = st;
}

Passing Side

1 Argument is passed
Passing a Student Object

```java
class LibraryCard {
    private Student owner;
    public void setOwner(Student st) {
        owner = st;
    }
}

LibraryCard card2;
card2 = new LibraryCard();
card2.setOwner(student);
```

Passing Side

Receiving Side

1 Argument is passed

*State of Memory*

- **Student**
  - name: "Jon Java"
  - email: "jj@javauniv.edu"

- **LibraryCard**
  - owner
  - borrowCnt: 0

1 Argument is passed
Passing a Student Object

```java
class LibraryCard {
    private Student owner;
    public void setOwner(Student st) {
        owner = st;
    }
}

LibraryCard card2;
card2 = new LibraryCard();
card2.setOwner(student);
```

Passing Side

Receiving Side

State of Memory

1. Argument is passed
Passing a Student Object

```java
class LibraryCard {
    private Student owner;
    public void setOwner(Student st) {
        owner = st;
    }
}

LibraryCard card2;
card2 = new LibraryCard();
card2.setOwner(student);
```

Passing Side

1. Argument is passed

Receiving Side

1. Argument is passed
2. Value is assigned to the data member

State of Memory

- **Student**
  - `name`: "Jon Java"
  - `email`: "jj@javauniv.edu"

- **LibraryCard**
  - `owner`
  - `borrowCnt`: 0
Passing a Student Object

```java
LibraryCard card2;
card2 = new LibraryCard();
card2.setOwner(student);
```

Passing Side

```java
class LibraryCard {
    private Student owner;
    public void setOwner(Student st) {
        owner = st;
    }
}
```

Receiving Side

1. Argument is passed
2. Value is assigned to the data member

State of Memory

- **Student**
  - name: "Jon Java"
  - email: "jj@javauniv.edu"
- **LibraryCard**
  - owner
  - borrowCnt: 0
- **LibraryCard**
  - st

Diagram:
- **Passing Side**
  - `card2 = new LibraryCard();`
  - `card2.setOwner(student);`
- **Receiving Side**
  - `class LibraryCard { private Student owner; public void setOwner(Student st) { owner = st; } }
  - Argument is passed
  - Value is assigned to the data member
Passing a Student Object

```java
class LibraryCard {
    private Student owner;
    public void setOwner(Student st) {
        owner = st;
    }
}

LibraryCard card2;
card2 = new LibraryCard();
card2.setOwner(student);
```

Passing Side

Receiving Side

1. Argument is passed
2. Value is assigned to the data member

State of Memory

<table>
<thead>
<tr>
<th>: LibraryCard</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
</tr>
<tr>
<td>borrowCnt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>: Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>email</td>
</tr>
</tbody>
</table>

“Jon Java”
“jj@javauniv.edu”
We pass the same Student object to card1 and card2

```java
Student student;
LibraryCard card1, card2;

student = new Student();
student.setName("Jon Java");
student.setEmail("jj@javauniv.edu");

card1 = new LibraryCard();
card1.setOwner(student);
card1.checkOut(3);

card2 = new LibraryCard();
card2.setOwner(student);
```
Sharing an Object

- Since we are actually passing a reference to the same object, it results in the owner of two LibraryCard objects pointing to the same Student object.
Returning Objects

- As with arguments passed to a method,
  - when a method returns a primitive data type, the value is returned.
  - when a method returns a class data type, the reference to an object is returned.

- For example,
  - A constructor returns the reference to the newly created object.
Information Hiding and Visibility Modifiers

- The modifiers `public` and `private` designate the accessibility of data members and methods.
- If a class component (data member or method) is declared private, other (client) classes cannot access it.
- If a class component is declared public, other (client) classes can access it.
- Internal details of a class are declared private and hidden from the clients. This is **information hiding**.
Encapsulation

- From the outside the object’s behavior is limited to the public methods.
- Internal data items are only accessible through methods (accessor/mutator methods).
- If class data are hidden, they can only be modified by class methods.
- This is called **encapsulation** -- greatly improves program design and reusability.
- Encapsulation limits what and how users can access -- avoids accidental modifications.
Accessibility Example

```java
class Service {
    public int memberOne;
    private int memberTwo;
    public void doOne() {
        ...
    }
    private void doTwo() {
        ...
    }
}

Service obj = new Service();
obj.memberOne = 10;
obj.memberTwo = 20;
obj.doOne();
obj.doTwo();
...
```

Client

Service
class Service {
    public int memberOne;
    private int memberTwo;
    public void doOne() {
        ...
    }
    private void doTwo() {
        ...
    }
}

Client

... Service obj = new Service();
    obj.memberOne = 10;
    obj.memberTwo = 20;
    obj.doOne();
    obj.doTwo();
    ...

Service
Accessibility Example

Client

```java
... Service obj = new Service();
    obj.memberOne = 10;
    obj.memberTwo = 20;
    obj.doOne();
    obj.doTwo();
...
```

Service

```java
class Service {
    public int memberOne;
    private int memberTwo;
    public void doOne() {
        ...
    }
    private void doTwo() {
        ...
    }
}
```
Accessibility Example

Client

```java
... Service obj = new Service();
obj.memberOne = 10;
obj.memberTwo = 20;
obj.doOne();
obj.doTwo();
...
```

Service

```java
class Service {
    public int memberOne;
    private int memberTwo;
    public void doOne() {
        ...
    }
    private void doTwo() {
        ...
    }
}
```
Accessibility Example

```java
class Service {
    public int memberOne;
    private int memberTwo;
    public void doOne() {
        ...
    }
    private void doTwo() {
        ...
    }
}
```

```java
Service obj = new Service();
obj.memberOne = 10;
obj.memberTwo = 20;
obj.doOne();
obj.doTwo();
...
```

Client

Service
Data Members Should be private

- Data members are the implementation details of the class, so they should be invisible to the clients. Declare them private.
- Exception: Constants can (should) be declared public if they are meant to be used directly by the outside methods.
Guideline for Visibility Modifiers

Guidelines in determining the visibility of data members and methods:

- Declare class and instance variables private.
- Declare class and instance methods private if they are used only by the other methods in the same class.
- Declare class constants public if you want to make their values directly readable by the client programs. If the class constants are used for internal purposes only, then declare them private.
Guideline for Visibility Modifiers

Guidelines in determining the visibility of data members and methods:

- Declare class and instance variables private.
- Declare class and instance methods private if they are used only by the other methods in the same class.
- Declare class constants public if you want to make their values directly readable by the client programs. If the class constants are used for internal purposes only, then declare them private.

Default is **public**!
Diagram Notation for Visibility

- balance
- ownerName

+ AccountVer2 (String, double)
+ add(double)
+ deduct(double)
+ getCurrentBalance()
+ getOwnerName()
+ setOwnerName(String)

public – plus symbol (+)
private – minus symbol (-)
In Chapter 3, we introduced the use of constants.

We illustrate the use of constants in programmer-defined service classes here.

Remember, the use of constants

- provides a meaningful description of what the values stand for. `number = UNDEFINED;` is more meaningful than `number = -1;`
- provides easier program maintenance. We only need to change the value in the constant declaration instead of locating all occurrences of the same value in the program code
A Sample Use of Constants

class Dice {
    private static final int MAX_NUMBER = 6;
    private static final int MIN_NUMBER = 1;
    private static final int NO_NUMBER = 0;

    private int number;

    public Dice() {
        number = NO_NUMBER;
    }

    // Rolls the dice
    public void roll() {
        number = (int) (Math.floor(Math.random() *
                               (MAX_NUMBER - MIN_NUMBER + 1)) + MIN_NUMBER);
    }

    // Returns the number on this dice
    public int getNumber() {
        return number;
    }
}
A new type of Dice

class Dice {
    private static final int MAX_NUMBER = 9;
    private static final int MIN_NUMBER = 3;
    private static final int NO_NUMBER = 0;

    private int number;

    public Dice() {
        number = NO_NUMBER;
    }

    // Rolls the dice
    public void roll() {
        number = (int) (Math.floor(Math.random() *
                              (MAX_NUMBER - MIN_NUMBER + 1)) + MIN_NUMBER);
    }

    // Returns the number on this dice
    public int getNumber() {
        return number;
    }
}
class Dice {

    private static final int MAX_NUMBER = 9;
    private static final int MIN_NUMBER = 3;
    private static final int NO_NUMBER = 0;

    private int number;

    public Dice() {
        number = NO_NUMBER;
    }

    // Rolls the dice
    public void roll() {
        number = (int) (Math.floor(Math.random() *
                                (MAX_NUMBER - MIN_NUMBER + 1)) + MIN_NUMBER);
    }

    // Returns the number on this dice
    public int getNumber() {
        return number;
    }
}

A new type of Dice

Only change needed!
Identifier types

- Identifiers can be declared almost anywhere in a program.
- There are three main types of declarations:
  - **Data members** of a class
    - Declared outside any method
    - Usually at the beginning of the class definition
  - As **formal parameters** of a method
  - Within a method -- **local variables**
Local Variables

Local variables are declared within a method declaration and used for temporary services, such as storing intermediate computation results.

```java
public double convert(int num) {
    double result;
    result = Math.sqrt(num * num);
    return result;
}
```
Local Variables

- Local variables are declared within a method declaration and used for temporary services, such as storing intermediate computation results.

```java
public double convert(int num) {
    double result;
    result = Math.sqrt(num * num);
    return result;
}
```
Identifier extent and scope

- Each identifier refers to a piece of memory.
- That piece is reserved upon declaration.
- The lifetime of this reservation is called the extent of the identifier.
- The ability to access this location from a given line of code is called scope.
- Important to understand both.
- Extent and scope depend upon the type of variable and its declaration.
Extent
Extent

Object data members
- created when an object is created (by `new`)
- destroyed when the object is garbage collected (no more references to it)
- must be unique for each class
Extent

- **Object data members**
  - created when an object is created (by `new`)
  - destroyed when the object is garbage collected (no more references to it)
  - must be unique for each class

- **Formal parameters**
  - created each time the method is called
  - destroyed when the method finishes execution
  - must be unique for each method
Extent

- **Object data members**
  - created when an object is created (by `new`)
  - destroyed when the object is garbage collected (no more references to it)
  - must be unique for each class

- **Formal parameters**
  - created each time the method is called
  - destroyed when the method finishes execution
  - must be unique for each method

- **Local variables**
  - created upon declaration
  - destroyed at end of block
  - must be unique for each block
Extent

- **Object data members**
  - created when an object is created (by `new`)
  - destroyed when the object is garbage collected (no more references to it)
  - must be unique for each class

- **Formal parameters**
  - created each time the method is called
  - destroyed when the method finishes execution
  - must be unique for each method

- **Local variables**
  - created upon declaration
  - destroyed at end of block
  - must be unique for each block

- **Limiting extent allows compilers to reuse space**
Local, Parameter & Data Member
An identifier appearing inside a method can be a local variable, a parameter, or a data member.
Local, Parameter & Data Member

- An identifier appearing inside a method can be a local variable, a parameter, or a data member.
- The rules are
Local, Parameter & Data Member

An identifier appearing inside a method can be a local variable, a parameter, or a data member.

The rules are

- If there’s a matching local variable declaration or a parameter, then the identifier refers to the local variable or the parameter.
Local, Parameter & Data Member

- An identifier appearing inside a method can be a local variable, a parameter, or a data member.

- The rules are
  - If there’s a matching local variable declaration or a parameter, then the identifier refers to the local variable or the parameter.
  - Otherwise, if there’s a matching data member declaration, then the identifier refers to the data member.
An identifier appearing inside a method can be a local variable, a parameter, or a data member.

The rules are

- If there's a matching local variable declaration or a parameter, then the identifier refers to the local variable or the parameter.
- Otherwise, if there's a matching data member declaration, then the identifier refers to the data member.
- Otherwise, it is an error because there's no matching declaration.
Sample Matching

class MusicCD {
    private String artist;
    private String title;
    private String id;

    public MusicCD(String name1, String name2) {
        String ident;
        artist = name1;
        title  = name2;
        ident  = artist.substring(0,2) + "-" + title.substring(0,9);
        id = ident;
    }
    ...
}

class MusicCD {

    private String artist;
    private String title;
    private String id;

    public MusicCD(String name1, String name2) {
        String ident;
        artist = name1;
        title = name2;
        ident = artist.substring(0,2) + "-" + title.substring(0,9);
        id = ident;
    }

    ...
Sample Matching

class MusicCD {

    private String artist;
    private String title;
    private String id;

    public MusicCD(String name1, String name2) {

        String ident;
        artist = name1;
        title  = name2;
        ident  = artist.substring(0,2) + "-" +
                    title.substring(0,9);
        id  = ident;
    }

    ...

}
Sample Matching

class MusicCD {

    private String artist;
    private String title;
    private String id;

    public MusicCD(String name1, String name2) {
        String ident;
        artist = name1;
        title  = name2;
        ident  = artist.substring(0,2) + "-" + title.substring(0,9);
        id  = ident;
    }

    ...
class MusicCD {

    private String artist;
    private String title;
    private String id;

    public MusicCD(String name1, String name2) {
        String ident;

        artist = name1;

        title  = name2;

        ident  = artist.substring(0,2) + "-" + title.substring(0,9);

        id  = ident;
    }

    ...

}
class MusicCD {

private String artist;
private String title;
private String id;

public MusicCD(String name1, String name2) {
    String ident;

    artist = name1;
    title = name2;

    ident = artist.substring(0,2) + "-" + title.substring(0,9);

    id = ident;
}

...
class MusicCD {

private String artist;
private String title;
private String id;

public MusicCD(String name1, String name2) {
    String ident;
    artist = name1;
    title = name2;
    ident = artist.substring(0,2) + "-" + title.substring(0,9);
    id = ident;
}
...
Sample Matching

class MusicCD {
    private String artist;
    private String title;
    private String id;

    public MusicCD(String name1, String name2) {
        String ident;
        artist = name1;
        title = name2;
        ident = artist.substring(0, 2) + "-" + title.substring(0, 9);
        id = ident;
    }
    ...
}
Sample Matching

```java
class MusicCD {
    private String artist;
    private String title;
    private String id;

    public MusicCD(String name1, String name2) {
        String ident;
        artist = name1;
        title = name2;
        ident = artist.substring(0,2) + "-" + title.substring(0,9);
        id = ident;
    }
    ...
}
```
```java
class MusicCD {
    private String artist;
    private String title;
    private String id;

    public MusicCD(String name1, String name2) {
        String ident;
        artist = name1;
        title = name2;
        ident = artist.substring(0,2) + "-" + title.substring(0,9);
        id = ident;
    }
    ...
}
```
class MusicCD {

    private String artist;
    private String title;
    private String id;

    public MusicCD(String name1, String name2) {
        String ident;
        artist = name1;
        title = name2;
        ident = artist.substring(0,2) + "-" + title.substring(0,9);
        id = ident;
    }

    ...
Sample Matching

class MusicCD {

    private String artist;
    private String title;
    private String id;

    public MusicCD(String name1, String name2) {
        String title, ident;
        artist = name1;
        title  = name2;
        ident  = artist.substring(0,2) + "-" +
                  title.substring(0,9);
        id  = ident;
    }

    ...

}
Sample Matching

class MusicCD {

    private String artist;
    private String title;
    private String id;

    public MusicCD(String name1, String name2) {
        String title, ident;
        artist = name1;
        title = name2;
        ident = artist.substring(0,2) + "-" +
                title.substring(0,9);
        id = ident;
    }
    ...
}

Sample Matching

class MusicCD {

    private String artist;  
    private String title;  
    private String id;  

    public MusicCD(String name1, String name2) {
        String title, ident;  
        artist = name1;  
        title = name2;    
        ident = artist.substring(0,2) + "-" + 
                    title.substring(0,9);  
        id = ident;  
    }  

    ...  
}

A local variable can be declared just about anywhere!

Its **scope** (the area of code from where it is visible) is limited to the enclosing braces.

Statements within a pair of braces are called a **block**.

Local variables are destroyed when the block finishes execution.

Data members of a class are declared outside any method. Their scope is determined by **public** and **private** modifiers.
Calling Methods of the Same Class
Calling Methods of the Same Class

- So far, we have been calling a method of another class (object).
Calling Methods of the Same Class

- So far, we have been calling a method of another class (object).
- It is possible to call method of a class from another method of the same class.
  - in this case, we simply refer to a method without dot notation
Calling Methods of the Same Class

- So far, we have been calling a method of another class (object).
- It is possible to call method of a class from another method of the same class.
  - in this case, we simply refer to a method without dot notation

```java
public void myMethod()
{
    AClass obj = new Bclass();
    obj.doWork();
}
```
Calling Methods of the Same Class

- So far, we have been calling a method of another class (object).
- It is possible to call method of a class from another method of the same class.
  - in this case, we simply refer to a method without dot notation

```java
public class AClass {
    public void myMethod() {
        BClass obj = new BClass();
        obj.doWork();
    }
}
```

```java
public class BClass {
    public void doWork() {
        ...
    }
    public void myMethod() {
        doWork();
    }
}
```

Dot notation necessary when calling method of another object.
So far, we have been calling a method of another class (object).

It is possible to call method of a class from another method of the same class.

- in this case, we simply refer to a method without dot notation

```java
public void myMethod()
{
    Bclass obj = new Bclass();
    obj.doWork();
}
```

Dot notation necessary when calling method of another object.
So far, we have been calling a method of another class (object).

It is possible to call method of a class from another method of the same class.

- In this case, we simply refer to a method without dot notation

```java
public void myMethod(){
    Bclass obj = new Bclass();
    obj.doWork();
}
```

Dot notation necessary when calling method of another object.

```java
public void doWork(){
    ...
}
```

Dot notation optional for calling method of the same class.

```java
public void myMethod(){
    doWork();
}
```
Any class can be set to be a main class.

All you have to do is to include the main method.
Any class can be set to be a main class. All you have to do is to include the main method.

class Bicycle {
    //definition of the class as shown before comes here
    //The main method that shows a sample
    //use of the Bicycle class
    public static void main(String[] args) {
        Bicycle myBike;
        myBike = new Bicycle();
        myBike.setOwnerName("Jon Java");
        System.out.println(myBike.getOwnerName() + "owns a bicycle");
    }
}
Problem statement:

*Write a loan calculator program that computes both monthly and total payments for a given loan amount, annual interest rate, and loan period.*
Overall Plan

Tasks:

- Get three input values: \texttt{loanAmount}, \texttt{interestRate}, and \texttt{loanPeriod}.
- Compute the monthly and total payments.
- Output the results.
Required Classes

LoanCalculator

JOptionPane
Loan
PrintStream

input
computation
output
Development Steps

We will develop this program in five steps:

1. Start with the main class LoanCalculator. Define a temporary placeholder Loan class.
2. Implement the input routine to accept three input values.
3. Implement the output routine to display the results.
4. Implement the computation routine to compute the monthly and total payments.
5. Finalize the program.
Step 1 Design

The methods of the LoanCalculator class

<table>
<thead>
<tr>
<th>Method</th>
<th>Visibility</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>public</td>
<td>Starts the loan calculation. Calls other methods</td>
</tr>
<tr>
<td>computePayment</td>
<td>private</td>
<td>Give three parameters, compute the monthly and total payments</td>
</tr>
<tr>
<td>describeProgram</td>
<td>private</td>
<td>Displays a short description of a program</td>
</tr>
<tr>
<td>displayOutput</td>
<td>private</td>
<td>Displays the output</td>
</tr>
<tr>
<td>getInput</td>
<td>private</td>
<td>Gets three input values</td>
</tr>
</tbody>
</table>
Step 1 Code

Directory: Chapter4/Step1

Source Files:
LoanCalculator.java
Loan.java
Step 1 Test

- In the testing phase, we run the program multiple times and verify that we get the following output

```plaintext
inside describeProgram
inside getInput
inside computePayment
inside displayOutput
```
Step 2 Design

- Design the input routines
  - LoanCalculator will handle the user interaction of prompting and getting three input values
  - LoanCalculator calls the setAmount, setRate and setPeriod of a Loan object.
Step 2 Code

Directory: Chapter4/Step2

Source Files:
LoanCalculator.java
Loan.java
Step 2 Test

- We run the program numerous times with different input values
- Check the correctness of input values by echo printing

```java
System.out.println("Loan Amount: $" + loan.getAmount());
System.out.println("Annual Interest Rate:" + loan.getRate() + "%");
System.out.println("Loan Period (years):" + loan.getPeriod());
```
Step 3 Design

- We will implement the `displayOutput` method.
- We will reuse the same design we adopted in Chapter 3 sample development.
Step 3 Design

- We will implement the `displayOutput` method.
- We will reuse the same design we adopted in Chapter 3 sample development.

<table>
<thead>
<tr>
<th>Monthly payment:</th>
<th>$143.47</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total payment:</td>
<td>$17216.50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loan Amount:</th>
<th>$10000.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annual Interest Rate:</td>
<td>12.0%</td>
</tr>
<tr>
<td>Loan Period (years):</td>
<td>10</td>
</tr>
<tr>
<td>Monthly payment is</td>
<td>$143.47</td>
</tr>
<tr>
<td>TOTAL payment is</td>
<td>$17216.50</td>
</tr>
</tbody>
</table>
Step 3 Code

Directory: Chapter4/Step3

Source Files:
 LoanCalculator.java
 Loan.java
Step 3 Test

- We run the program numerous times with different input values and check the output display format.
- Adjust the formatting as appropriate
Step 4 Design

- Two methods getMonthlyPayment and getTotalPayment are defined for the Loan class.
- We will implement them so that they work independent of each other.
- It is considered a poor design if the clients must call getMonthlyPayment before calling getTotalPayment.
Step 4 Code

Directory:  Chapter4/Step4

Source Files:
  LoanCalculator.java
  Loan.java
Step 4 Test

- We run the program numerous times with different types of input values and check the results.

<table>
<thead>
<tr>
<th>Loan Amount</th>
<th>Annual Interest Rate</th>
<th>Loan Period (in Years)</th>
<th>Monthly Payment</th>
<th>Total Payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>10</td>
<td>10</td>
<td>132.151</td>
<td>15858.088</td>
</tr>
<tr>
<td>15000</td>
<td>7</td>
<td>15</td>
<td>134.824</td>
<td>24268.363</td>
</tr>
<tr>
<td>10000</td>
<td>12</td>
<td>10</td>
<td>143.471</td>
<td>17216.514</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>5</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>30</td>
<td>8.5</td>
<td>50</td>
<td>0.216</td>
<td>129.373</td>
</tr>
</tbody>
</table>
Step 5: Finalize

- We implement the `describeProgram` method.
- We will format the monthly and total payments to two decimal places using DecimalFormat.

Directory: Chapter4/Step5

Source Files (final version):
- LoanCalculator.java
- Loan.java