Week 2 Java Basics (1/2)

4009 DATA STRUCTURES & ALGORITHM (DSA)

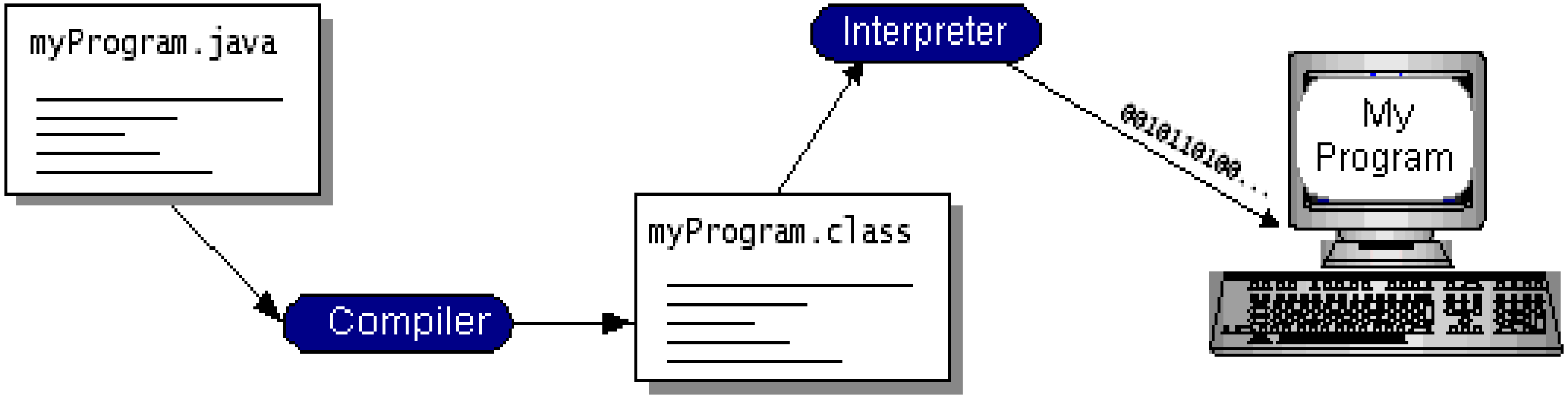
JAVA BASICS PART 1 OF 2

Learning Outcomes

- ❖Understand and discuss the basics of Java programming
- ❖Identify and discuss a Java program structure such as the main method, statement blocks and variables.
- ❖Define, modify and use Java classes, objects and methods.
- ❖Differentiate between objects, classes, methods, parameters and constructors.
- ❖Understand the use of modifiers with Java Methods, classes and variables.
- ❖Implement a single Java program with using relevant class name, objects, variables and comments

Content Overview

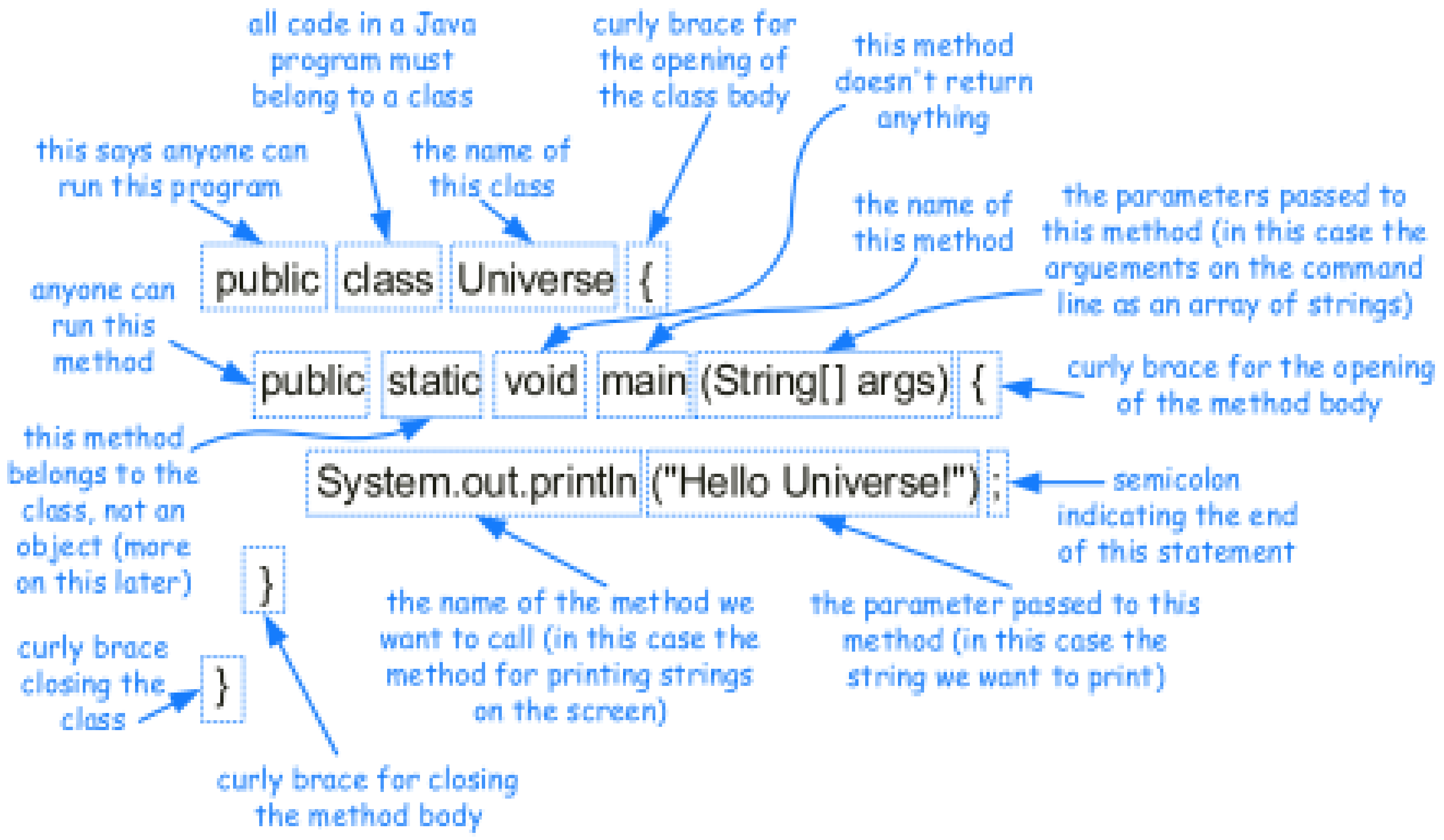
Basics of compiled and interpreted



File types you must come across with Java.

- **javac -** compiler,
- **java -** interpreter,
- **javadoc** – generator of API documentation,
- **appletviewer** – applet browser,
- **jar** – tool for jar files
- **jdb -** debuggger,

Getting Started with Java



```
public class Universe {
public static void main (String[] args) {
System.out.println ("Hello Universe!");
}
}
```

all code in a Java program must belong to a class
curly brace for the opening of the class body
this method doesn't return anything
this says anyone can run this program
the name of this class
the name of this method
the parameters passed to this method (in this case the arguements on the command line as an array of strings)
anyone can run this method
curly brace for the opening of the method body
this method belongs to the class, not an object (more on this later)
semicolon indicating the end of this statement
the name of the method we want to call (in this case the method for printing strings on the screen)
the parameter passed to this method (in this case the string we want to print)
curly brace closing the class
curly brace for closing the method body

Figure 1.1: A "Hello Universe!" program.

Objects

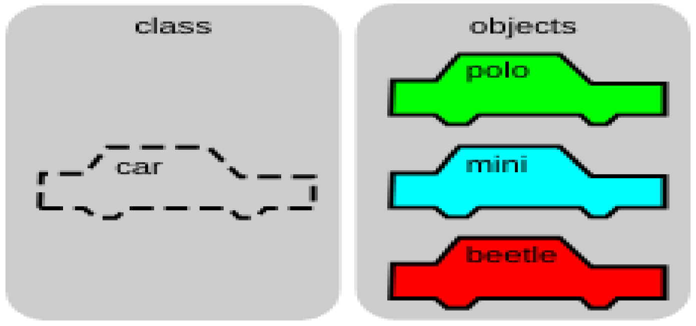
Objects in programming is solution to problems with procedural languages such as Basic, Fortran, Cobol, C or Pascal.

Learn in OOP that an object contains both methods and variables.

For example: a thermostat object contain methods ***furnace_on()*** and ***furnace_off()*** methods with variables called ***currentTemp*** and ***desiredTemp***.

The new entity (the object) solves several problems simultaneously in programming.

Object in a program corresponds more closely to an object in the real world



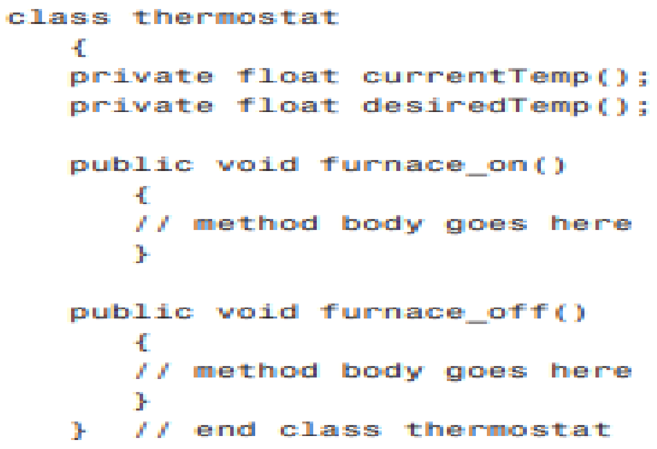
Example 2: class is car that has objects polo, mini & beetle

In programming term – ***every object is an instance of a class. Therefore, polo, mini and beetle are instances of a car***

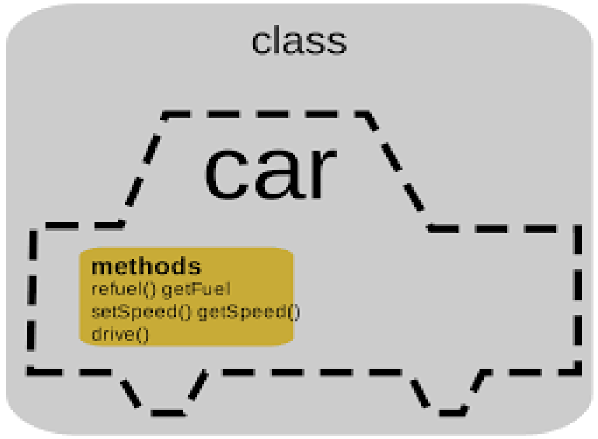
Classes

A class is a specification—a blueprint—for one or more objects.

Here's how a ***thermostat class***, for example, might look in Java or Example 2: diagram show ***car class*** with methods



```
class thermostat
   {
   private float currentTemp();
   private float desiredTemp();

   public void furnace_on()
      {
      // method body goes here
      }

   public void furnace_off()
      {
      // method body goes here
      }
   }  // end class thermostat
```



The Java keyword **class** introduces the class specification, followed by the **name** you want to give the class; here it's ***thermostat***. **Enclosed in curly brackets** are **the fields and methods** that **make up the class**. We've left out the bodies of the methods; normally, each would have many lines of program code.

Idea of objects

A programming unit which has associated:

- **Variables (data), and**
- **Methods (functions to manipulate this data).**

Of the many problems of other programming languages, objective oriented programing like Java addresses two problem below.

- Poor Real-World Modeling
- Crude Organization

Example of a class specifying objects.

```
class thermostat {
  private float currentTemp;
  private float desiredTemp;

  public void furnaceOn() {
    …
  }
```

Creating Objects

Specifying a class doesn’t create any objects of that class

To actually create objects in Java, you must use the keyword **new**.

At the same time an object is created, you need to store a reference to it in a variable of suitable type—that is, the same type as the class.

What’s a reference?

Think of a reference as a name for an object.

For example, we would create two references to type thermostat, create two new thermostat objects, and store references to them in these variables:

```
thermostat therm1, therm2;      // create two references
therm1 = new thermostat();      // create two objects and
therm2 = new thermostat();      // store references to them
```

Incidentally, creating an object is also called ***instantiating it***, and an object is often referred to as an ***instance*** of a class.

Accessing Object Methods

After you specify a class and create some objects of that class, other parts of your program need to interact with these objects.

Typically, other parts of the program interact with an object's methods, not with its data (fields).

For example, to tell the therm2 object to turn on the furnace:

```
therm2.furnace_on();
```

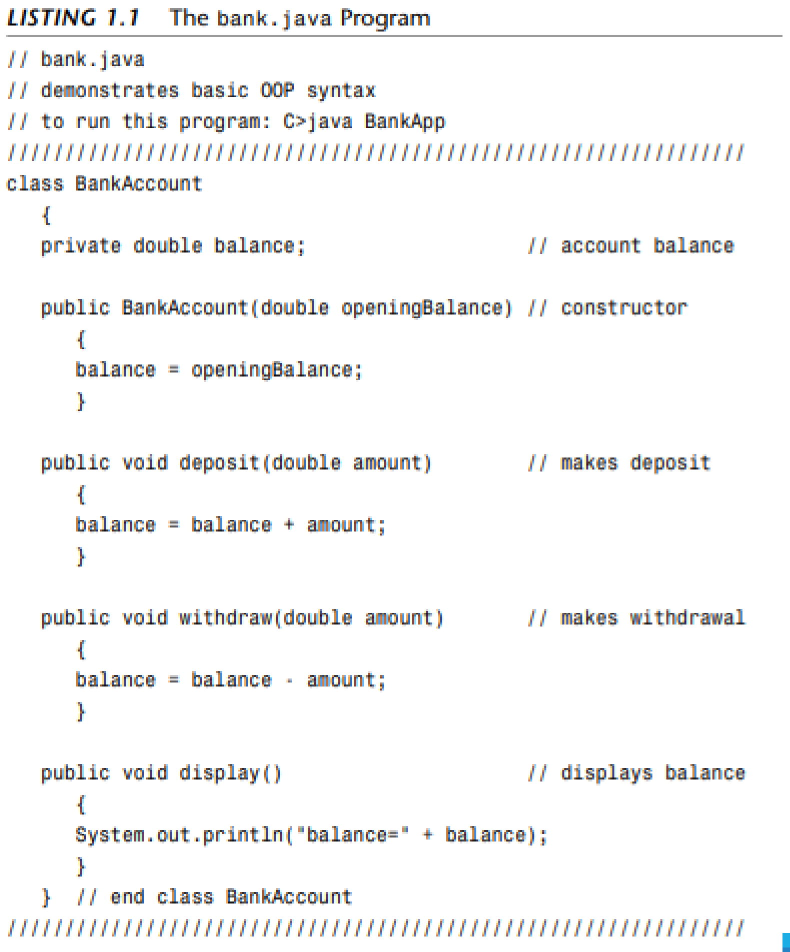
The dot operator (.) associates an object with one of its.

To summarize:

- ***Objects contain both methods and fields (data).***
- ***A class is a specification for any number of objects.***
- ***To create an object, you use the keyword new in conjunction with the class name.***
- ***To invoke a method for a particular object, you use the dot operator.***

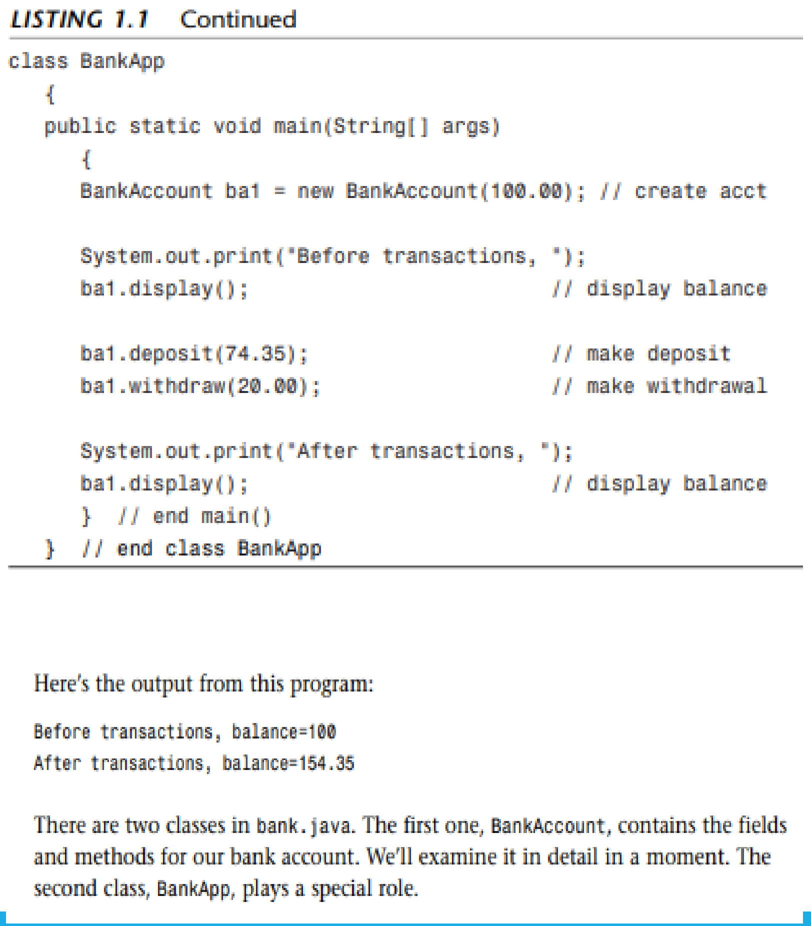
A runnable Object-Oriented Program

Let’s look at an object-oriented program that runs and generates actual output. It features a class called BankAccount that models a checking account at a bank. The program creates an account with an opening balance, displays the balance, makes a deposit and a withdrawal, and then displays the new balance.



LISTING 1.1 The `bank.java` Program

```
// bank.java
// demonstrates basic OOP syntax
// to run this program: C>java BankApp
////////////////////////////////////////////////////////////////
class BankAccount
   {
   private double balance;                     // account balance

   public BankAccount(double openingBalance) // constructor
      {
      balance = openingBalance;
      }

   public void deposit(double amount)        // makes deposit
      {
      balance = balance + amount;
      }

   public void withdraw(double amount)       // makes withdrawal
      {
      balance = balance - amount;
      }

   public void display()                     // displays balance
      {
      System.out.println("balance=" + balance);
      }
   }  // end class BankAccount
////////////////////////////////////////////////////////////////
```



LISTING 1.1 Continued

```
class BankApp
   {
   public static void main(String[] args)
      {
      BankAccount ba1 = new BankAccount(100.00); // create acct

      System.out.print("Before transactions, ");
      ba1.display();                         // display balance

      ba1.deposit(74.35);                    // make deposit
      ba1.withdraw(20.00);                   // make withdrawal

      System.out.print("After transactions, ");
      ba1.display();                         // display balance
      }  // end main()
   }  // end class BankApp
```

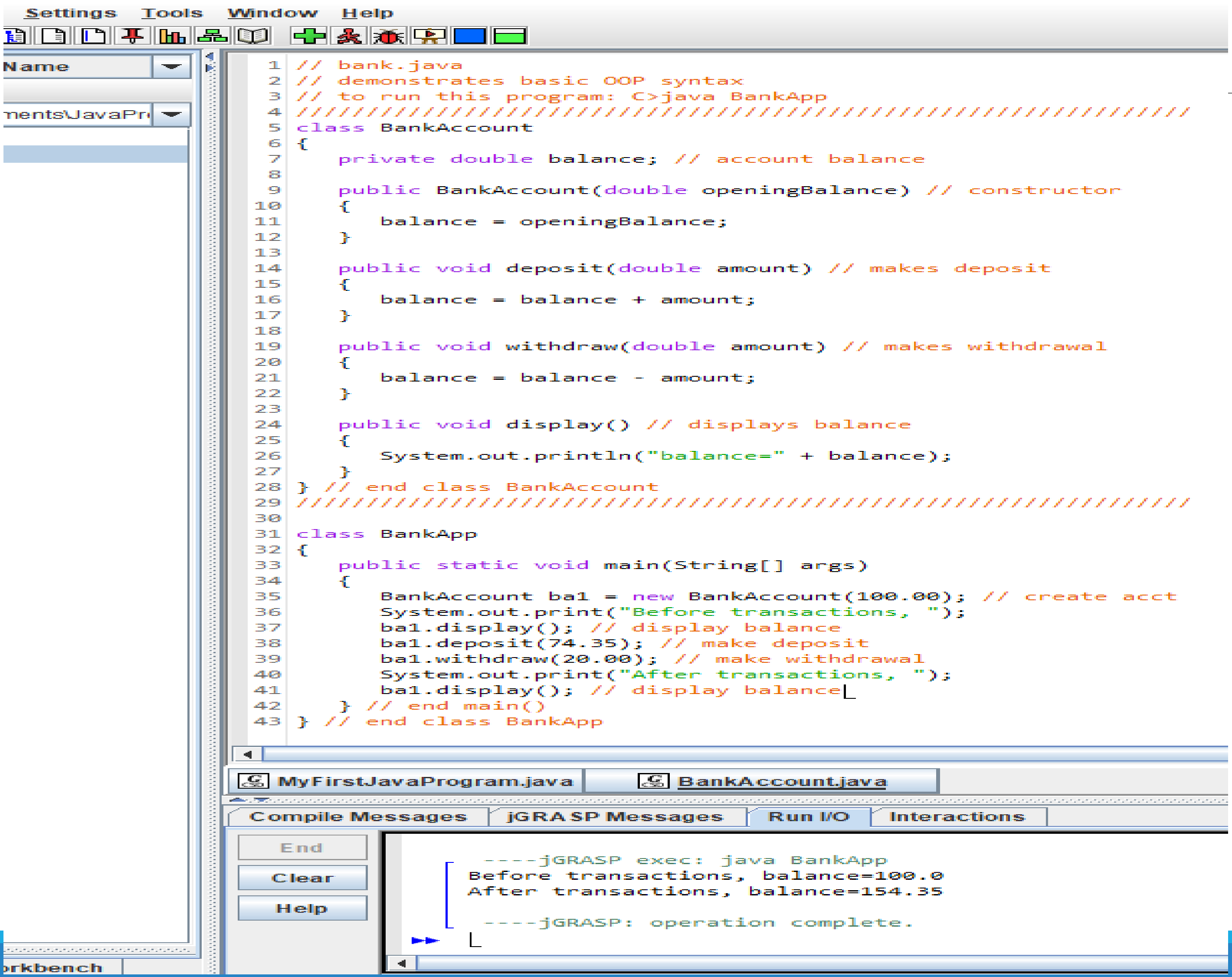
Here’s the output from this program:

```
Before transactions, balance=100
After transactions, balance=154.35
```

There are two classes in `bank.java`. The first one, `BankAccount`, contains the fields and methods for our bank account. We’ll examine it in detail in a moment. The second class, `BankApp`, plays a special role.

A runnable Object-Oriented Program

Creation of BankAccount.java program in jGrasp



```java
// bank.java
// demonstrates basic OOP syntax
// to run this program: C>java BankApp
///////////////////////////////////////////////////////////////
class BankAccount
{
   private double balance; // account balance

   public BankAccount(double openingBalance) // constructor
   {
      balance = openingBalance;
   }

   public void deposit(double amount) // makes deposit
   {
      balance = balance + amount;
   }

   public void withdraw(double amount) // makes withdrawal
   {
      balance = balance - amount;
   }

   public void display() // displays balance
   {
      System.out.println("balance=" + balance);
   }
} // end class BankAccount
///////////////////////////////////////////////////////////////

class BankApp
{
   public static void main(String[] args)
   {
      BankAccount ba1 = new BankAccount(100.00); // create acct
      System.out.print("Before transactions, ");
      ba1.display(); // display balance
      ba1.deposit(74.35); // make deposit
      ba1.withdraw(20.00); // make withdrawal
      System.out.print("After transactions, ");
      ba1.display(); // display balance
   } // end main()
} // end class BankApp
```

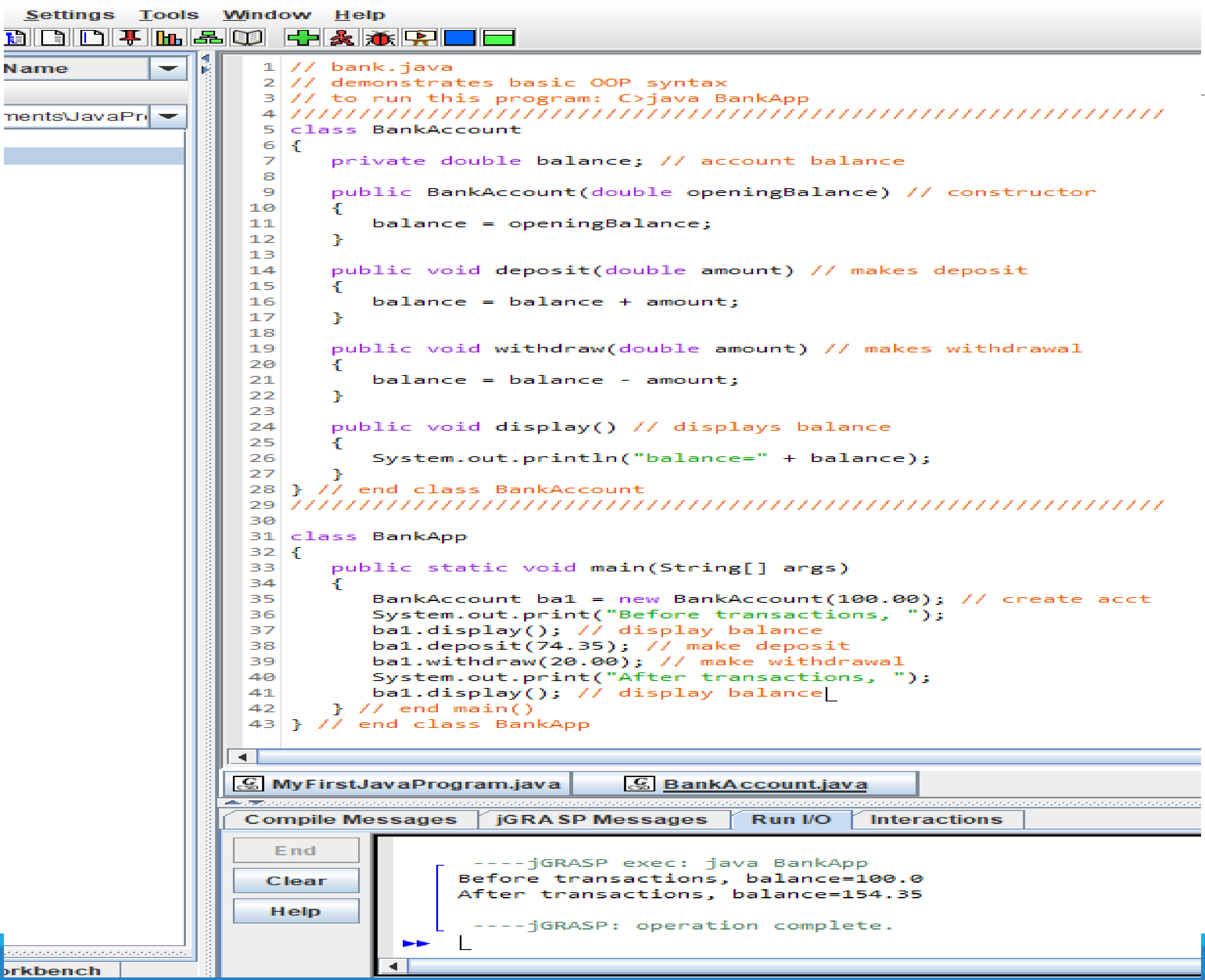
```
 ----jGRASP exec: java BankApp
Before transactions, balance=100.0
After transactions, balance=154.35

 ----jGRASP: operation complete.
```

BankAccount Class

The only data field in the BankAccount class is the amount of money in the account, called balance. There are three methods. The deposit() method adds an amount to the balance, withdrawal() subtracts an amount, and display() displays the balance.

A runnable Object-Oriented Program

Creation of BankAccount.java program in jGrasp



```java
// bank.java
// demonstrates basic OOP syntax
// to run this program: C>java BankApp
////////////////////////////////////////////////////////////////
class BankAccount
{
   private double balance; // account balance

   public BankAccount(double openingBalance) // constructor
   {
      balance = openingBalance;
   }

   public void deposit(double amount) // makes deposit
   {
      balance = balance + amount;
   }

   public void withdraw(double amount) // makes withdrawal
   {
      balance = balance - amount;
   }

   public void display() // displays balance
   {
      System.out.println("balance=" + balance);
   }
} // end class BankAccount
////////////////////////////////////////////////////////////////

class BankApp
{
   public static void main(String[] args)
   {
      BankAccount ba1 = new BankAccount(100.00); // create acct
      System.out.print("Before transactions, ");
      ba1.display(); // display balance
      ba1.deposit(74.35); // make deposit
      ba1.withdraw(20.00); // make withdrawal
      System.out.print("After transactions, ");
      ba1.display(); // display balance
   } // end main()
} // end class BankApp
```

```
 ----jGRASP exec: java BankApp
Before transactions, balance=100.0
After transactions, balance=154.35

 ----jGRASP: operation complete.
```

BankApp Class

Every Java application must have a main() method; execution of the program starts at the beginning of main().

The main() method creates an object of class BankAccount, initialized to a value of 100.00, which is the opening balance, with this statement:
BankAccount ba1 = new BankAccount(100.00); // create acct

The System.out.print() method displays the string used as its argument, Before transactions:, and the account displays its balance with this statement:
ba1.display();

The program then makes a deposit to, and a withdrawal from, the account:
ba1.deposit(74.35);
ba1.withdraw(20.00);

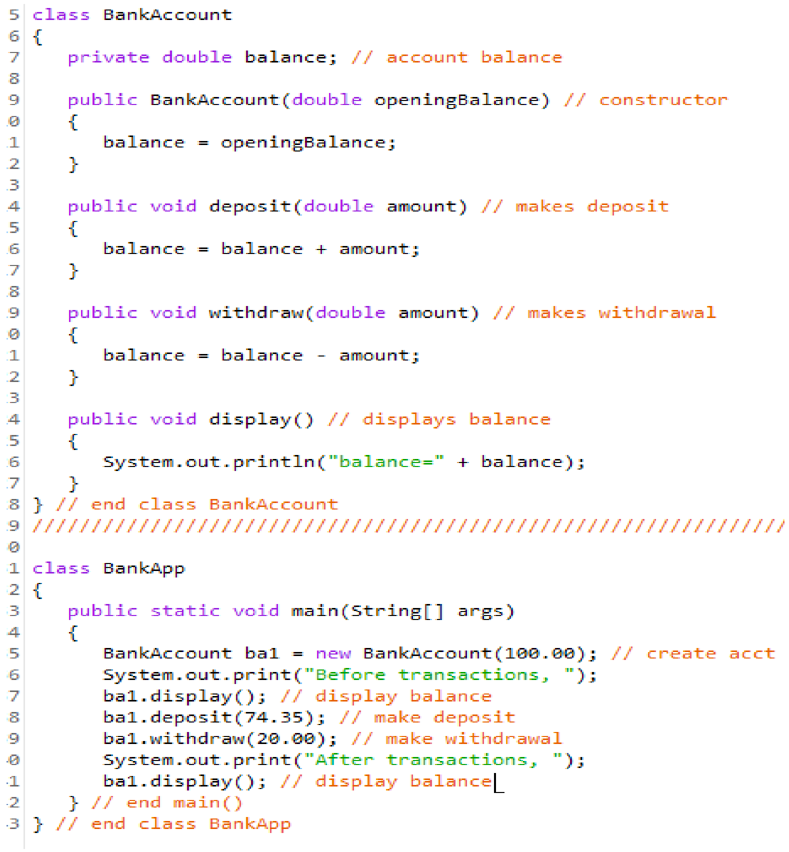
Finally, the program displays the new account balance and terminates

The BankAccount class explained

The only data field in the BankAccount class is the amount of money in the account, called ***balance***.

There are **three methods**.

The ***deposit()*** method adds an amount to the balance, ***withdrawal()*** subtracts an amount, and ***display()*** displays the balance



```
class BankAccount
{
   private double balance; // account balance

   public BankAccount(double openingBalance) // constructor
   {
      balance = openingBalance;
   }

   public void deposit(double amount) // makes deposit
   {
      balance = balance + amount;
   }

   public void withdraw(double amount) // makes withdrawal
   {
      balance = balance - amount;
   }

   public void display() // displays balance
   {
      System.out.println("balance=" + balance);
   }
} // end class BankAccount
////////////////////////////////////////////////////////////////////////////////

class BankApp
{
   public static void main(String[] args)
   {
      BankAccount ba1 = new BankAccount(100.00); // create acct
      System.out.print("Before transactions, ");
      ba1.display(); // display balance
      ba1.deposit(74.35); // make deposit
      ba1.withdraw(20.00); // make withdrawal
      System.out.print("After transactions, ");
      ba1.display(); // display balance
   } // end main()
} // end class BankApp
```

Class Modifiers

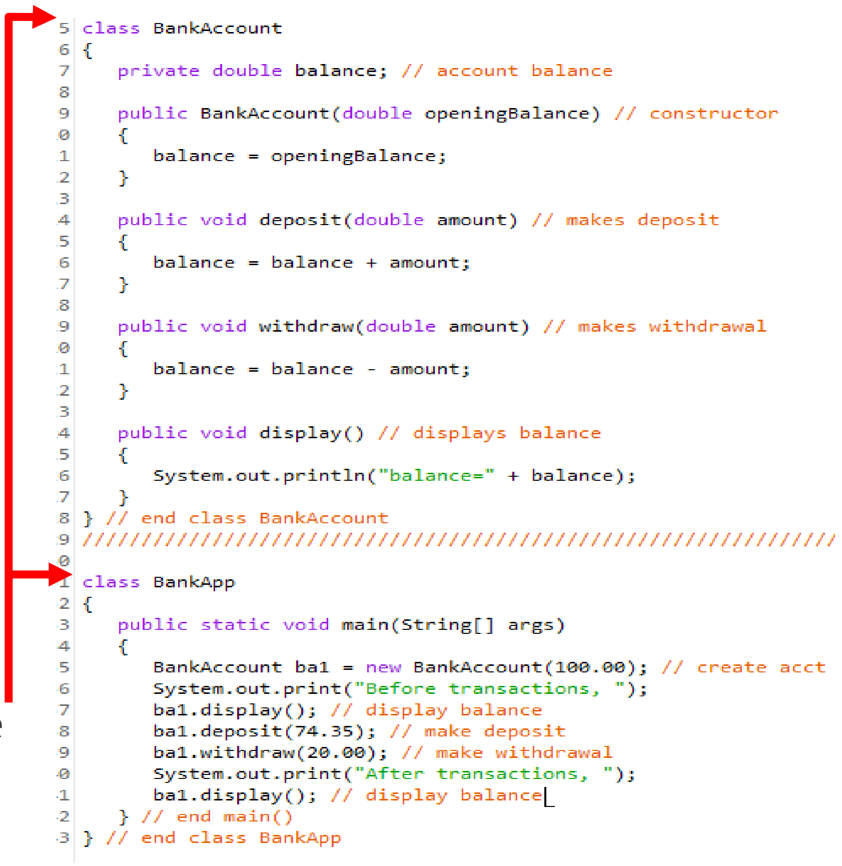
Class modifiers which are optional keywords precede the class keyword.

The public class modifier describes a class that can be instantiated or extended by anything in the same package or anything that imports the class.

The final class modifier describes a class that can have no subclasses

The Abstract class modifier describes a class that has abstract methods. Abstract methods are declared with the abstract keyword and are empty.

Note: if the public class modifier is not used, the class is considered friendly. Meaning it can be used and instantiated by all classes in the same package.



```
class BankAccount
{
   private double balance; // account balance

   public BankAccount(double openingBalance) // constructor
   {
      balance = openingBalance;
   }

   public void deposit(double amount) // makes deposit
   {
      balance = balance + amount;
   }

   public void withdraw(double amount) // makes withdrawal
   {
      balance = balance - amount;
   }

   public void display() // displays balance
   {
      System.out.println("balance=" + balance);
   }
} // end class BankAccount
////////////////////////////////////////////////////////////////

class BankApp
{
   public static void main(String[] args)
   {
      BankAccount ba1 = new BankAccount(100.00); // create acct
      System.out.print("Before transactions, ");
      ba1.display(); // display balance
      ba1.deposit(74.35); // make deposit
      ba1.withdraw(20.00); // make withdrawal
      System.out.print("After transactions, ");
      ba1.display(); // display balance
   } // end main()
} // end class BankApp
```

Base types

The types of objects are determined by the class the come from.

Java provides the following base types (also called ***primitive types***)

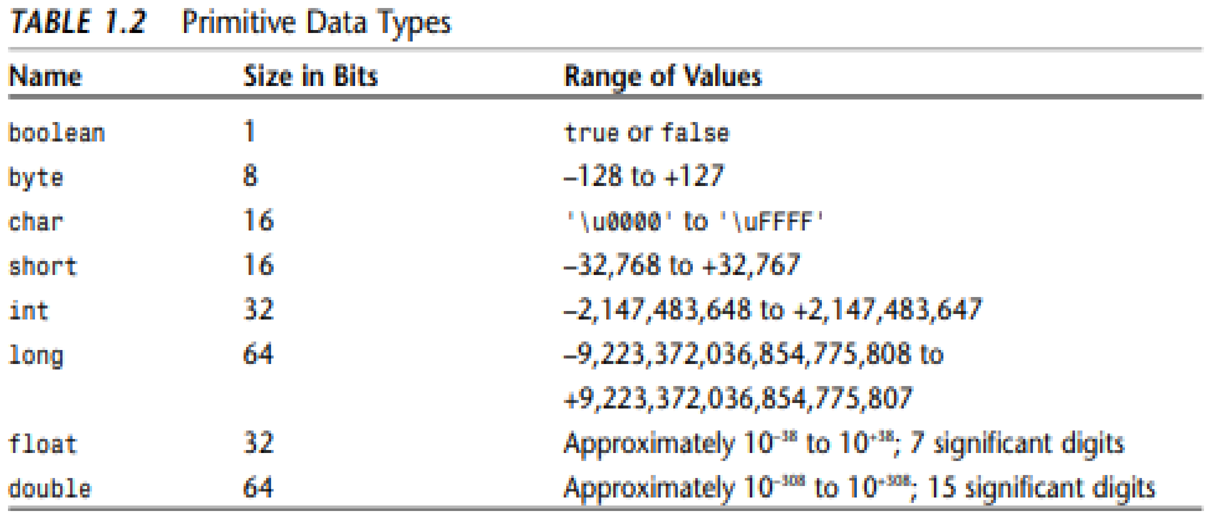
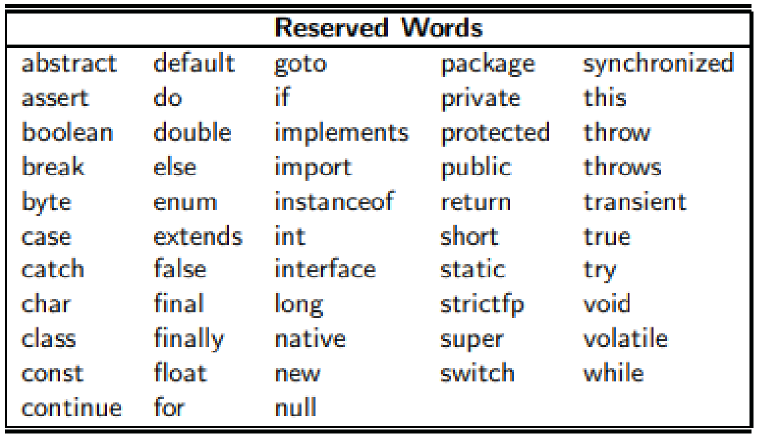


TABLE 1.2 Primitive Data Types

Name	Size in Bits	Range of Values
`boolean`	1	`true` or `false`
`byte`	8	–128 to +127
`char`	16	`'\u0000'` to `'\uFFFF'`
`short`	16	–32,768 to +32,767
`int`	32	–2,147,483,648 to +2,147,483,647
`long`	64	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
`float`	32	Approximately 10^{-38} to 10^{+38}; 7 significant digits
`double`	64	Approximately 10^{-308} to 10^{+308}; 15 significant digits

Other example Week 1 – slide 25

Reserve words & Comments



Reserved Words				
abstract	default	goto	package	synchronized
assert	do	if	private	this
boolean	double	implements	protected	throw
break	else	import	public	throws
byte	enum	instanceof	return	transient
case	extends	int	short	true
catch	false	interface	static	try
char	final	long	strictfp	void
class	finally	native	super	volatile
const	float	new	switch	while
continue	for	null		

Table 1.1: A listing of the reserved words in Java. These names cannot be used as class, method, or variable names.

Comments

Java allows a programmer to embed comments, which are annotations provided for human readers that are not processed by the Java compiler.

Java allows two kinds of comments: inline comments and block comments.

Java uses a "//" to begin an inline comment, ignoring everything subsequently on that line.

For example: // This is an inline comment.

While inline comments are limited to one line, Java allows multiline comments in the form of block comments. J

ava uses a "/*" to begin a block comment and a "*/" to close it.

For example: /* * This is a block comment. */

Block comments that begin with "/**" (note the second asterisk) have a special purpose, allowing a program, called Javadoc, to read these comments and automatically generate software documentation.

Object References

Creating a new object involves the use of the **new** operator to allocate the object's memory space and use the object's constructor to initialize this space.

The location (address) of this space is then assigned to a **reference** variable.

- The reference variable can be viewed as a "pointer" to some object.
- The reference variable can ask the object to perform an action or can access the objects data.

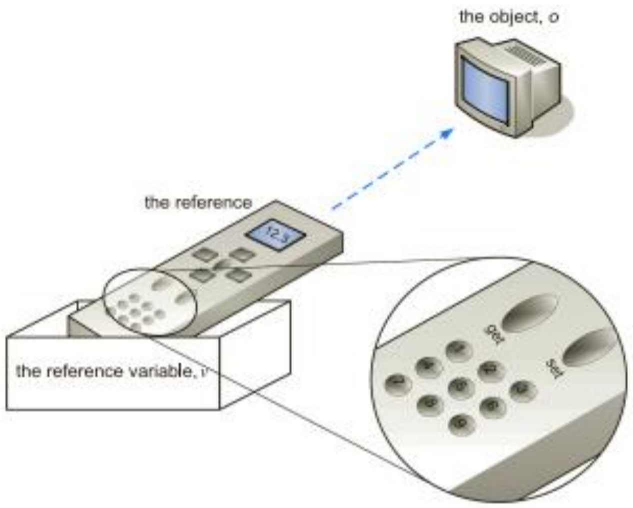


Figure 1.2: Illustrating the relationship between objects and object reference variables. When we assign an object reference (that is, memory address) to a reference variable, it is as if we are storing that object's remote control at that variable.

Modifiers

Immediately before the definition of a class, instance variable, or method in Java, keywords known as ***modifiers*** can be placed to convey additional stipulations about that definition.

Access Control Modifiers – control the level of access that the defining class grants to other classes in the Java program.

Public	Anyone can access public instance variables.
Protected	Only methods of the same package or of its subclasses can access protected instance variables.
Private	Only methods of the same class (not methods of a subclass) can access private instance.

Modifiers

The static modifier – declared for any variable or method of a class.

When a variable of a class is declared as static, its value is associated with the class as a whole, rather than with each individual instance of that class. Static variables are used to store “global” information about a class.

When a method of a class is declared as static, it too is associated with the class itself, and not with a particular instance of the class.

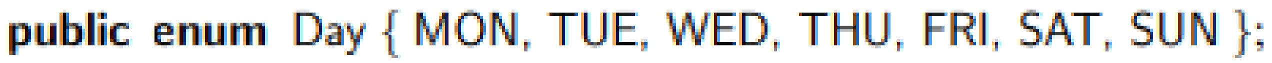
The final modifier – initialised as part of that declaration but can never again be assigned a new value.

Enum Types

Java supports enumerate types called **enums.**

These are types that are only allowed to take on values that come from a specified set of names.

Modifier **enum** name {value_name0, value_name1, value_numN-1};



```
public enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
```

Methods

Methods in Java are chunks of code that can be called on a particular object.

2nd definition: A **method** is a block of code which only runs when it is called.

Methods can **accept parameters as arguments** and their behavior depends on the object they belong to and the values of any parameters that are passed.

Every method in Java is specified in the body of some class.

A method definition has two parts:

1. The signature which defines the parameters for a method
2. The body which defines what the method does or performs.

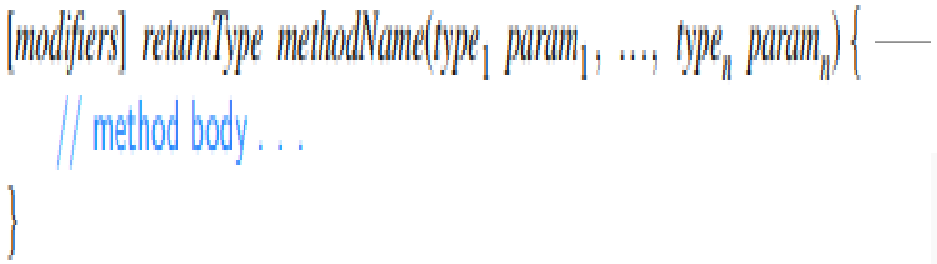
For example:

```
Public class main{
        static void myMethod(){
        }
}
```

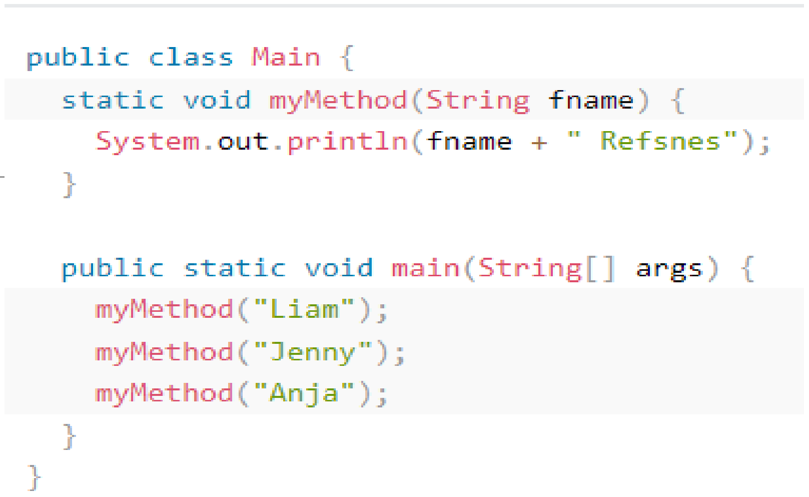
A method allows a programmer to send a message to an object. The method signature specifies how such a message should look and the method body specifies what the object will do when it received such as message.

Declaring Methods & method modifiers

Syntax for defining a method is as follows:



```
[modifiers] returnType methodName(type1 param1, ..., typen paramn) {
    // method body . . .
}
```



```
public class Main {
  static void myMethod(String fname) {
    System.out.println(fname + " Refsnes");
  }

  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
```

Method modifiers - can restrict the scope of a method.

Public	Anyone can access public instance methods
Protected	Only methods of the same package or of its subclasses can access protected methods.
Private	Only methods of the same class (not methods of a subclass) can access private methods.
Final	This is a method that cannot be overridden by a subclass.
Static	This is a method that is associated with the class itself and not with a particular instance of the class.

Parameters

A method’s parameters are defined in a comma-separated list enclosed in parentheses after the name of the method.

A parameter consists of two parts

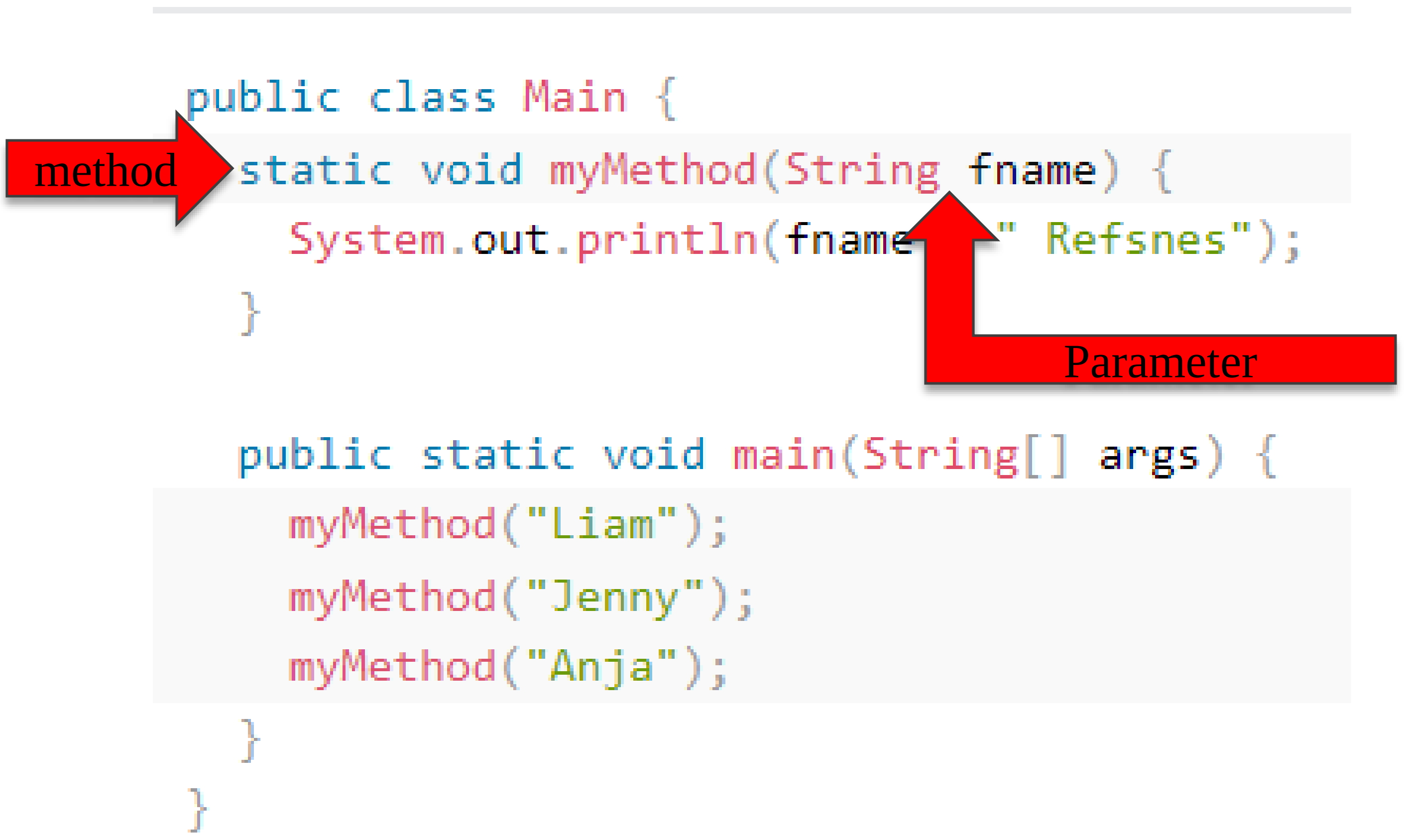
1. the parameter type.
2. the parameter name.

If a method has no parameters, then only an empty pair of parentheses is used.

All parameters in Java are passed by value, that is, any time we pass a parameter to a method, a copy of that parameter is made for use within the method body.

So if we pass an int variable to a method, then that variable’s integer value is copied. The method can change the copy but not the original. If we pass an object reference as a parameter to a method, then the reference is copied as well. Remember that we can have many different variables that all refer to the same object. Reassigning the internal reference variable inside a method will not change the reference that was passed in.

Method and Parameter



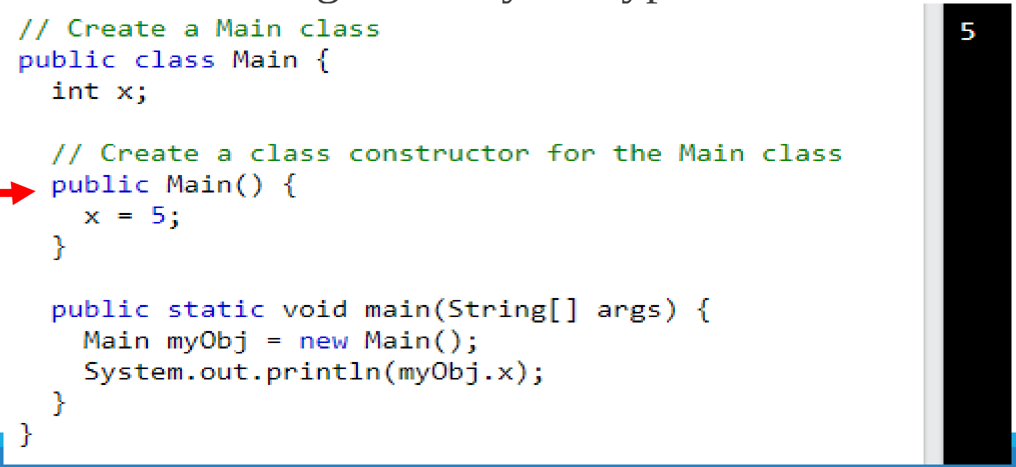
```
public class Main {
  static void myMethod(String fname) {
    System.out.println(fname     " Refsnes");
  }

  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
```

Constructors

A **constructor** is a special kind of method that is used to initialize a newly created instance of the class so that it will be in a consistent and stable initial state. This is typically achieved by initializing each instance variable of the object (unless the default value will suffice), although a constructor can perform more complex computation.

modifiers name($type_0$ $parameter_0$, ..., $type_{n-1}$ $parameter_{n-1}$) {
    // constructor body . . .
}

A class can have many constructors, but each must have a different signature, that is, each must be distinguished by the type and number of the parameters it takes.

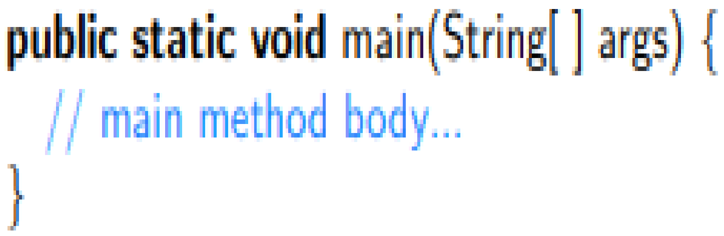


```java
// Create a Main class
public class Main {
  int x;

  // Create a class constructor for the Main class
  public Main() {
    x = 5;
  }

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

```
5
```

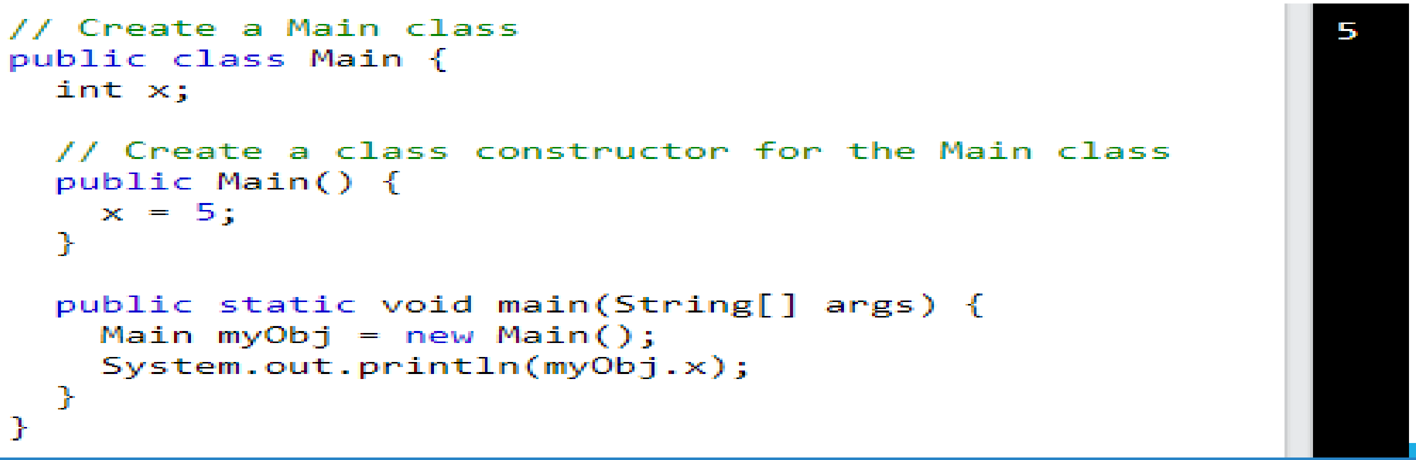
The Main Method

The primary control for an application in Java must begin in some class with the execution of a special method named main. This method must be declared as follows:



```java
public static void main(String[ ] args) {
  // main method body...
}
```

The args parameter is an array of String objects, that is, a collection of indexed strings, with the first string being args[0], the second being args[1], and so on.



```java
// Create a Main class
public class Main {
  int x;

  // Create a class constructor for the Main class
  public Main() {
    x = 5;
  }

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

```
5
```

Statement Blocks & Local Variables

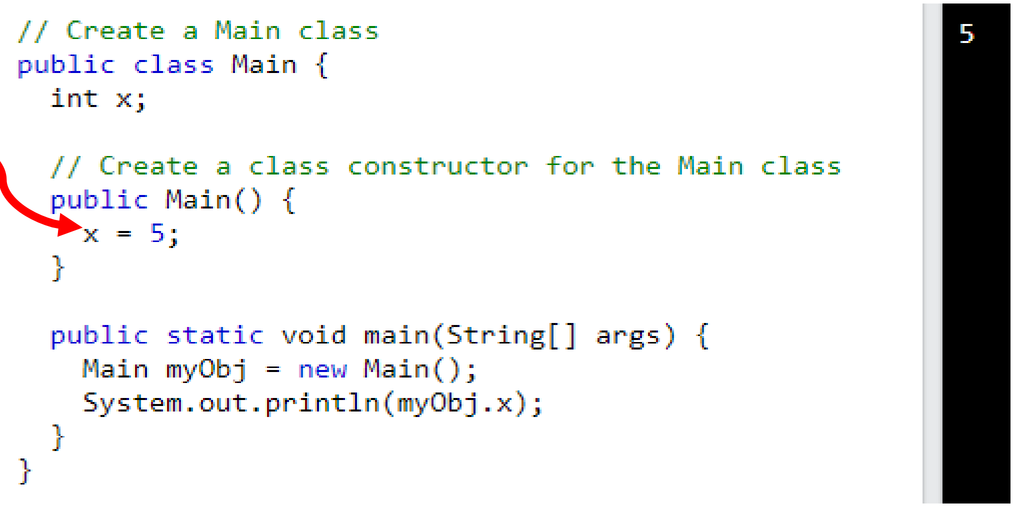
The body of a method is a ***statement block*** which is a sequence of statements and declarations to be performed between the **braces "{" and "}".**

Method bodies and other statement blocks can themselves have statement blocks nested inside of them.

In addition to statements that perform some action like calling the method of some object, statement blocks can contain declarations of ***local variables.***

There are two ways of declaring local variables:

1. *type name;*
2. *type name = initial_value;*



```java
// Create a Main class
public class Main {
  int x;

  // Create a class constructor for the Main class
  public Main() {
    x = 5;
  }

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

```
5
```

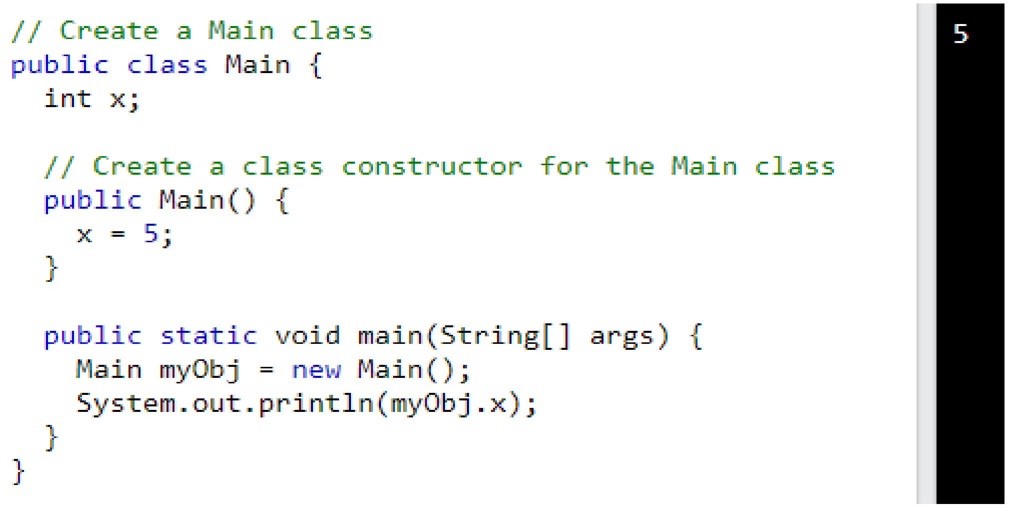
Statement Blocks & Local Variables

Every Java program has a **class name** which **must match** the **filename of the program**. ***Example public class Main{} is the class, therefore, the program file is called Main.java.***

The **curly braces {}** marks the **beginning and the end of a block of code**.

System is a **built-in Java class** that contains useful members such as out, which is short for **“output”.**

The **println() method** is short for **“print line”** used to print a value to the screen (or a file).



```java
// Create a Main class
public class Main {
  int x;

  // Create a class constructor for the Main class
  public Main() {
    x = 5;
  }

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

```
5
```

Topic next week

Week 3 Java Basics (Part 2 or 2)

DSA Week 2 activities

Refer to print out materials.

- This week, you are required to complete the questions and two labs.
  - Refer to the print out, answer all week 2 questions.
  - Refer to the print outs, complete the two labs activities using the lab computers.

Note: You can complete the activities in any order, however, make afford to complete and understand everything which prepares you for well for test 1, test 2, Major Assignment, Mid Semester Exam & Final Exam.