Introduction to Object Oriented Programming and Getting Started with Java

Core Java Day 1

Agenda

Brighter Blue

Day 01: Introduction to OOP and Getting Started with Java

01 Introduction to Object Oriented Programming (OOP)

02 Getting Started with Java

03 Basic Lexical Elements in Java

04 Primitive Data Types and Literals in Java

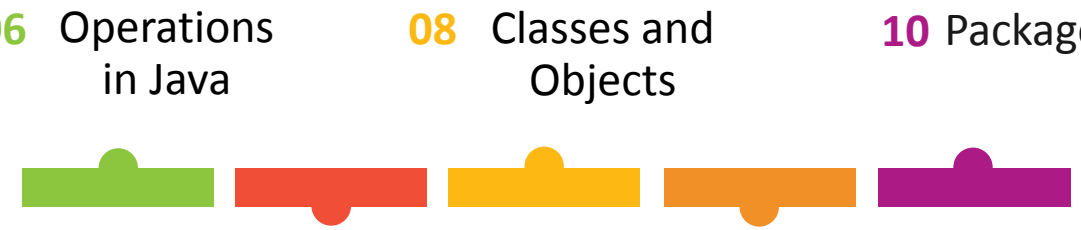
05 Strings and Arrays in Java

IBM

Agenda

Brighter Blue

Day 01: Introduction to OOP and Getting Started with Java



06 Operations in Java

08 Classes and Objects

10 Packages

07 Expressions and Control Flow in Java

09 Abstract Classes and Interfaces

Agenda

Brighter Blue

Day 02:
Advanced OOP with Examples

01 Abstraction and Encapsulation

03 Polymorphism

02 Inheritance

Agenda

Brighter Blue

Day 03: Core Java Threads and Garbage Collection

01 Java Threads

02 Garbage Collection

IBM

Agenda

Brighter Blue

Day 04: Core Java Input/Output, Exception and Error Handling

01 Input and Output Streams

03 Exception and Error Handling

02 File I/O (Input /Output) and NIO 2.0

Agenda

Brighter Blue

Day 05: Core Java Collection

01 Collections

02 Generics and Annotations

Agenda

Brighter Blue

Day 06: Core Java JDBC

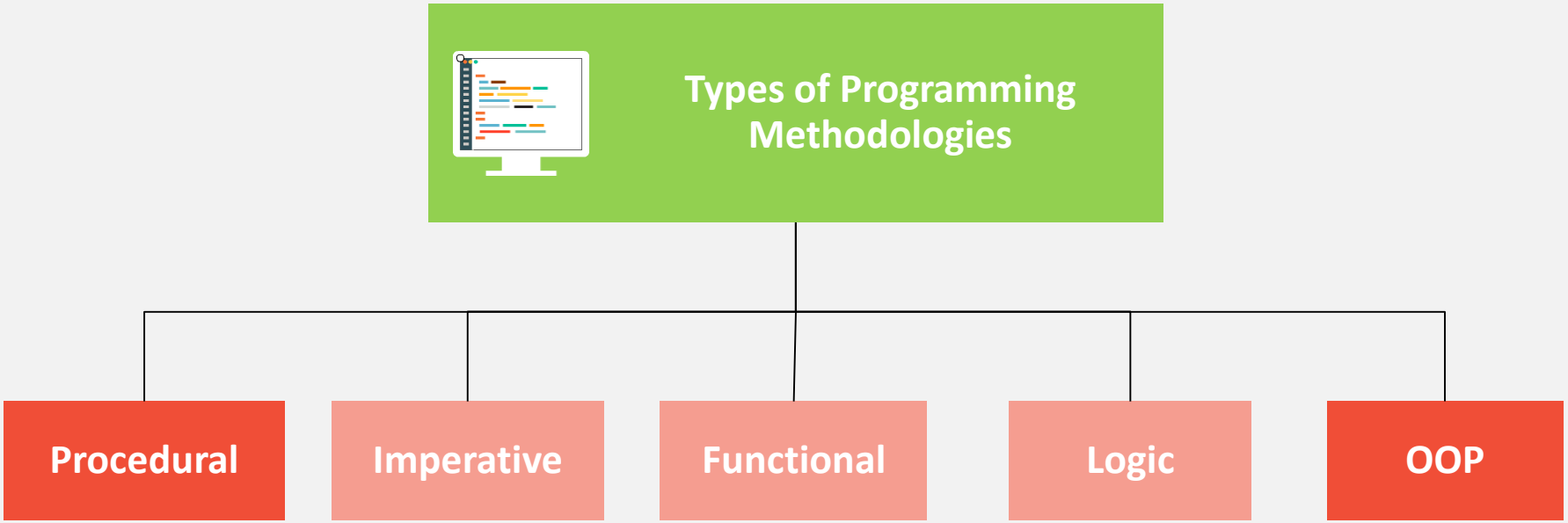
01 Database Architectures, JDBC Architecture, Drivers and API

Brighter Blue

01 Introduction to OOP

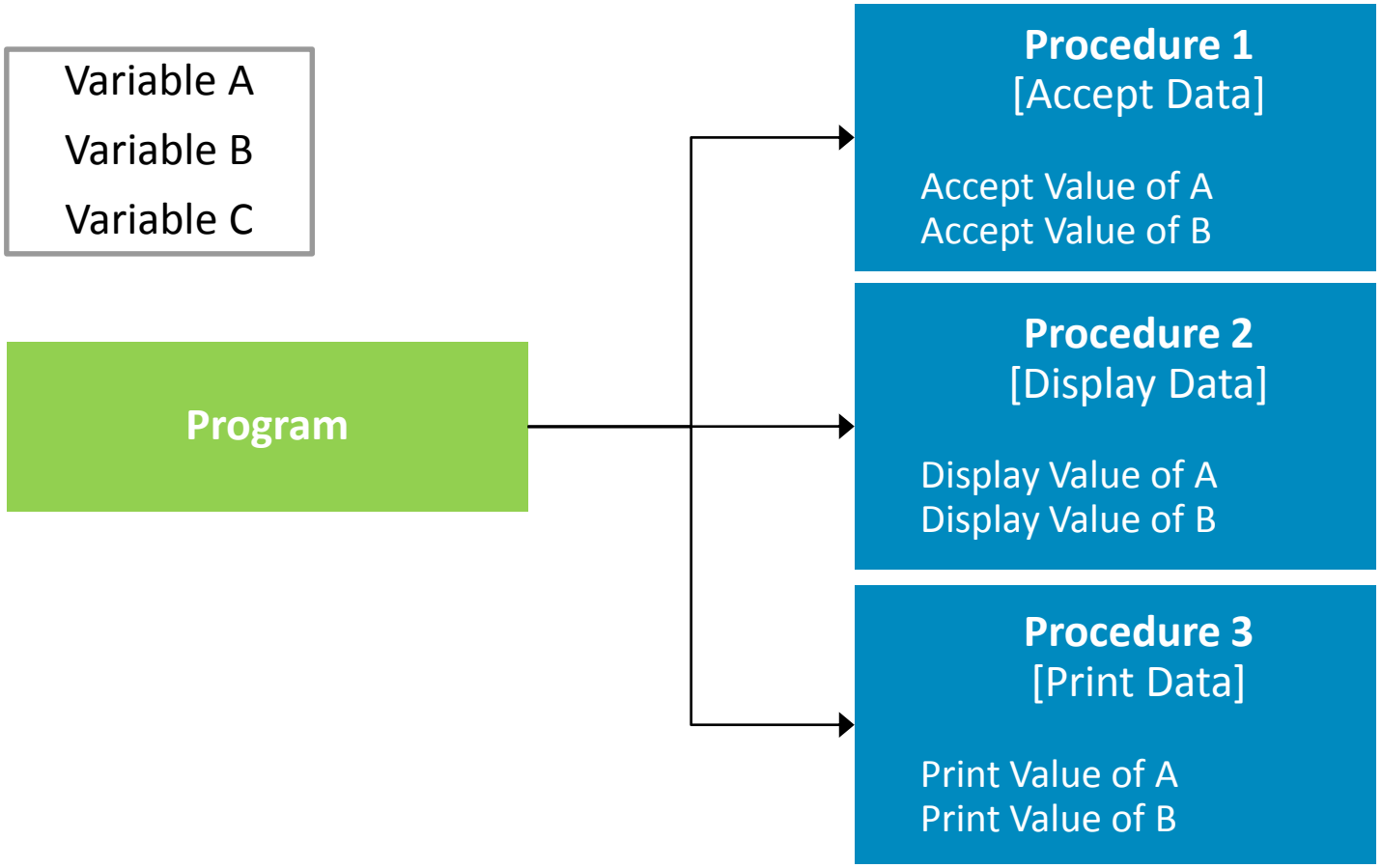
IBM

Types of **programming methodologies**



What is **procedural programming**?

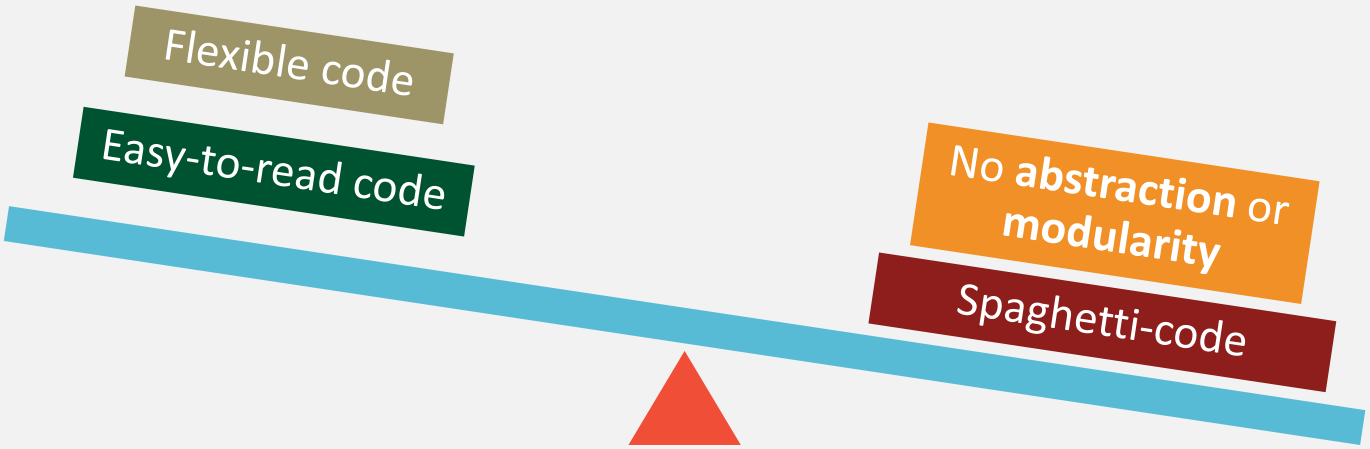
A large program is divided into a set of sub procedures or sub programs that perform specific tasks.



Benefits & limitations of procedural programming

Brighter Blue

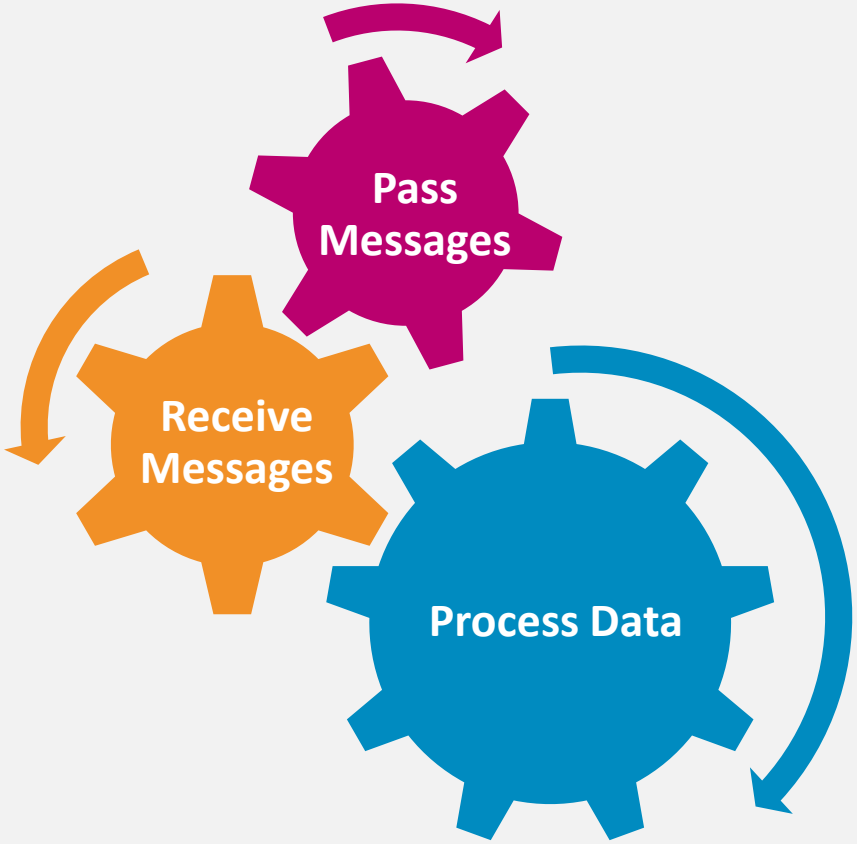
Benefits when developing **simple** applications



Limitations when developing **complex** applications

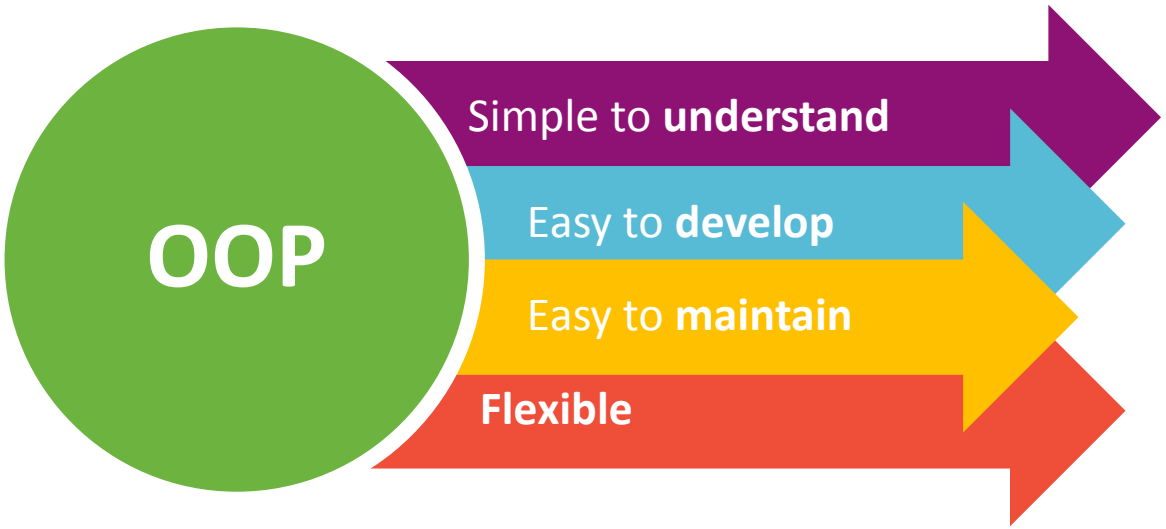
Procedural programing is better than sequential programing if you are developing simple applications. When the application is more complex, it becomes unmanageable.

What is **OOP**?



An object-oriented application uses a collection of objects, which communicate by passing messages to request services.

Aims and advantages of OOP

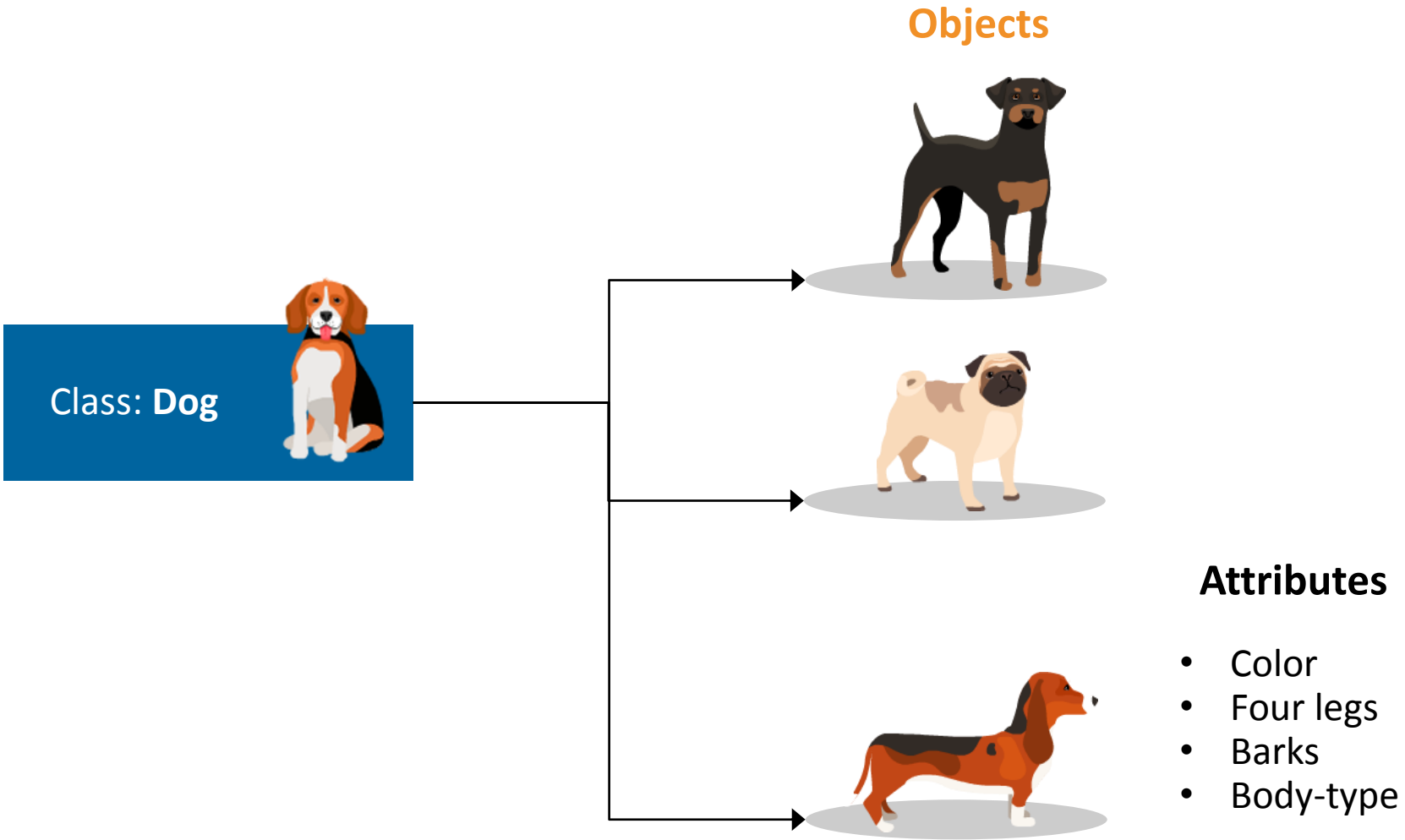


Advantages of OOP

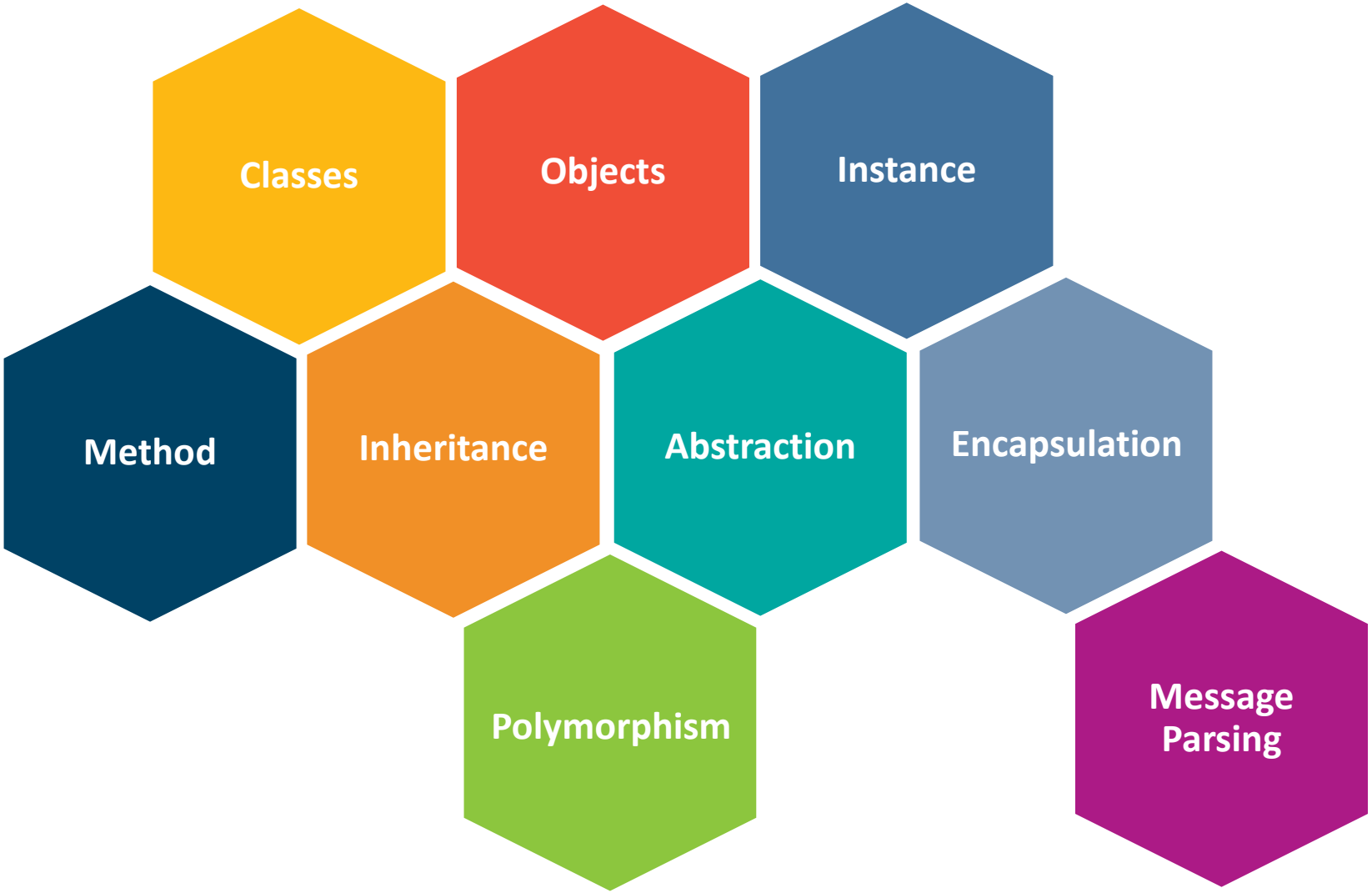
- Real-world programming
- Re-usability of code
- Modularity of code
- Resilience to change
- Information hiding

OOP models **real-world objects**

Brighter Blue



Fundamental concepts of Java



What are **classes** and **objects**?

Brighter Blue

State / Data

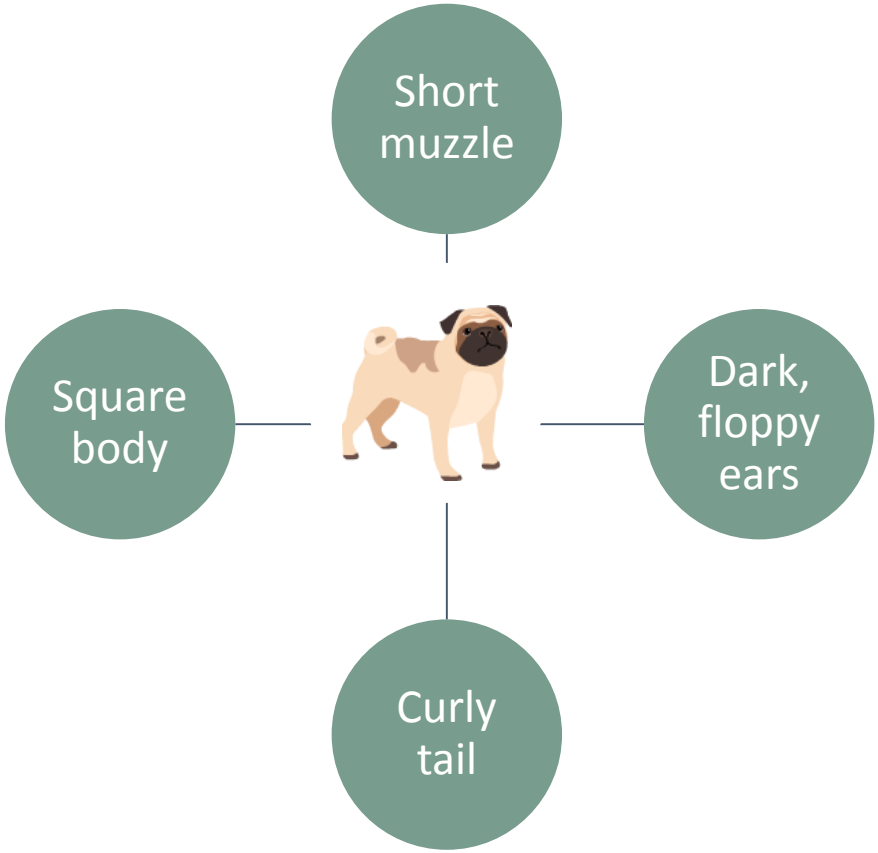
Behavior / Methods

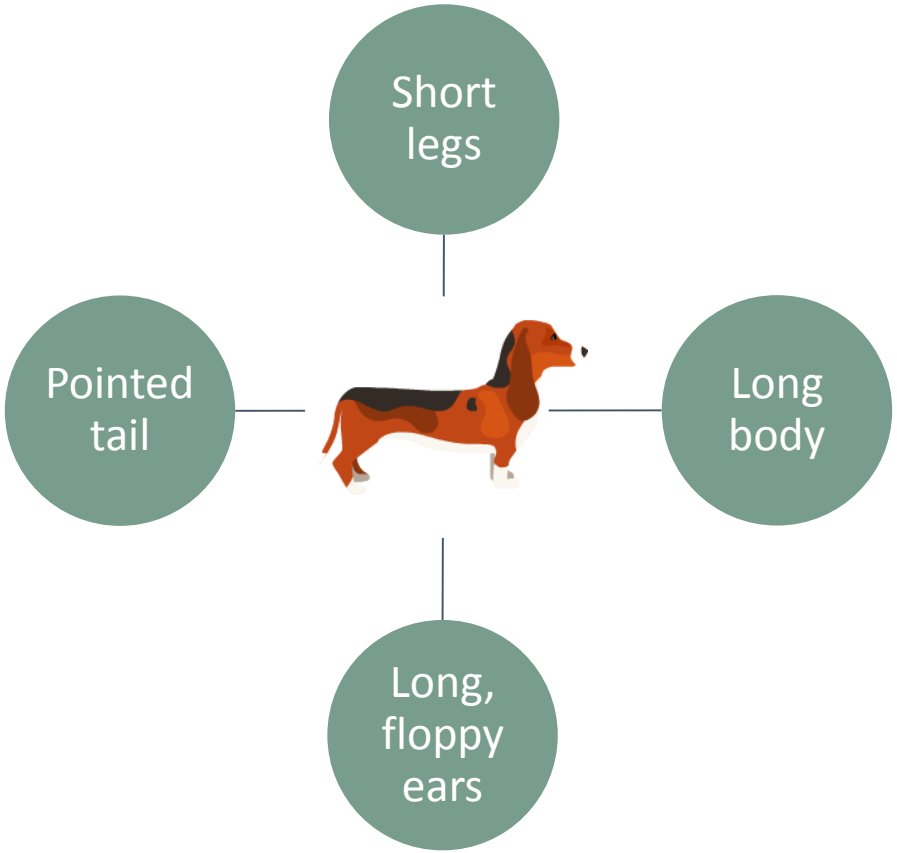
Class: **Dog**

Objects

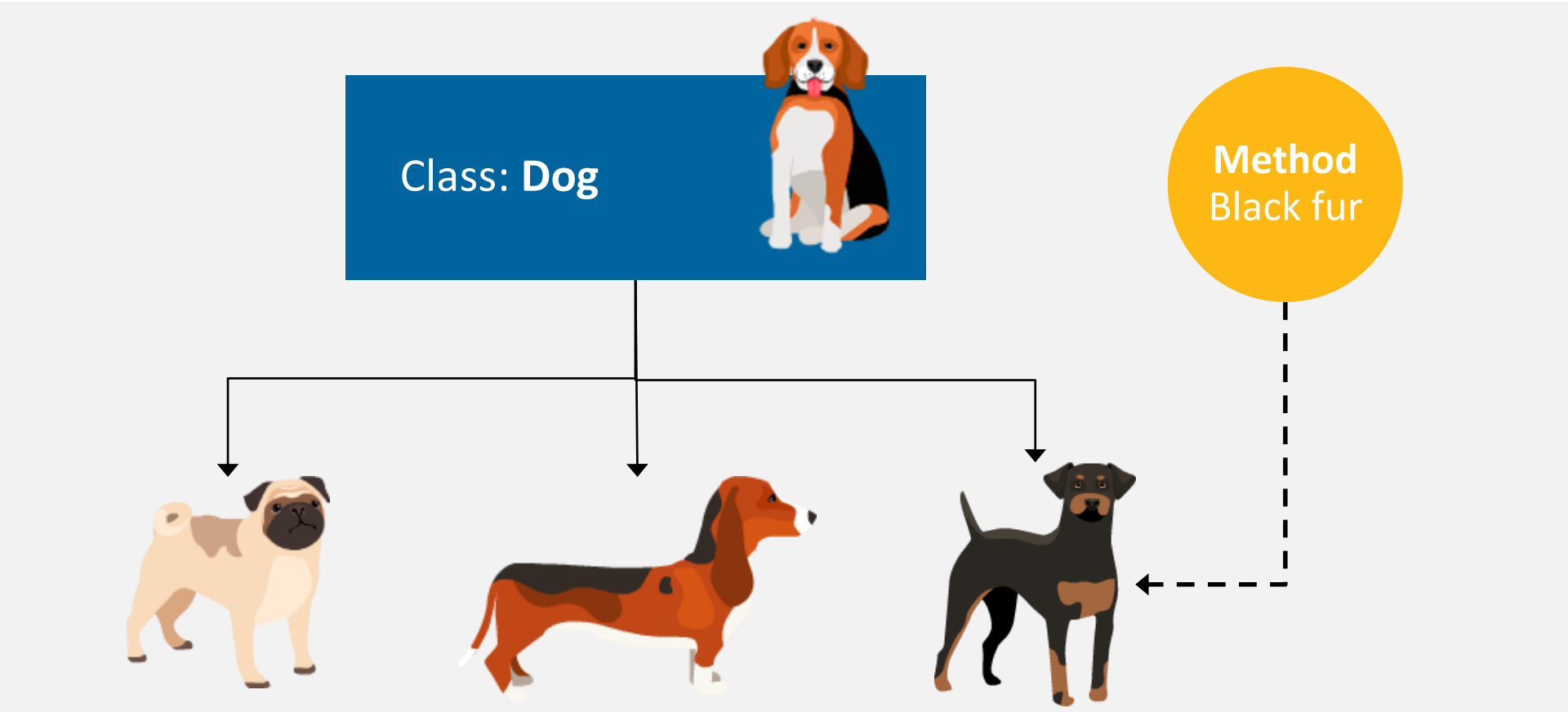
IBM

What is **abstraction**?

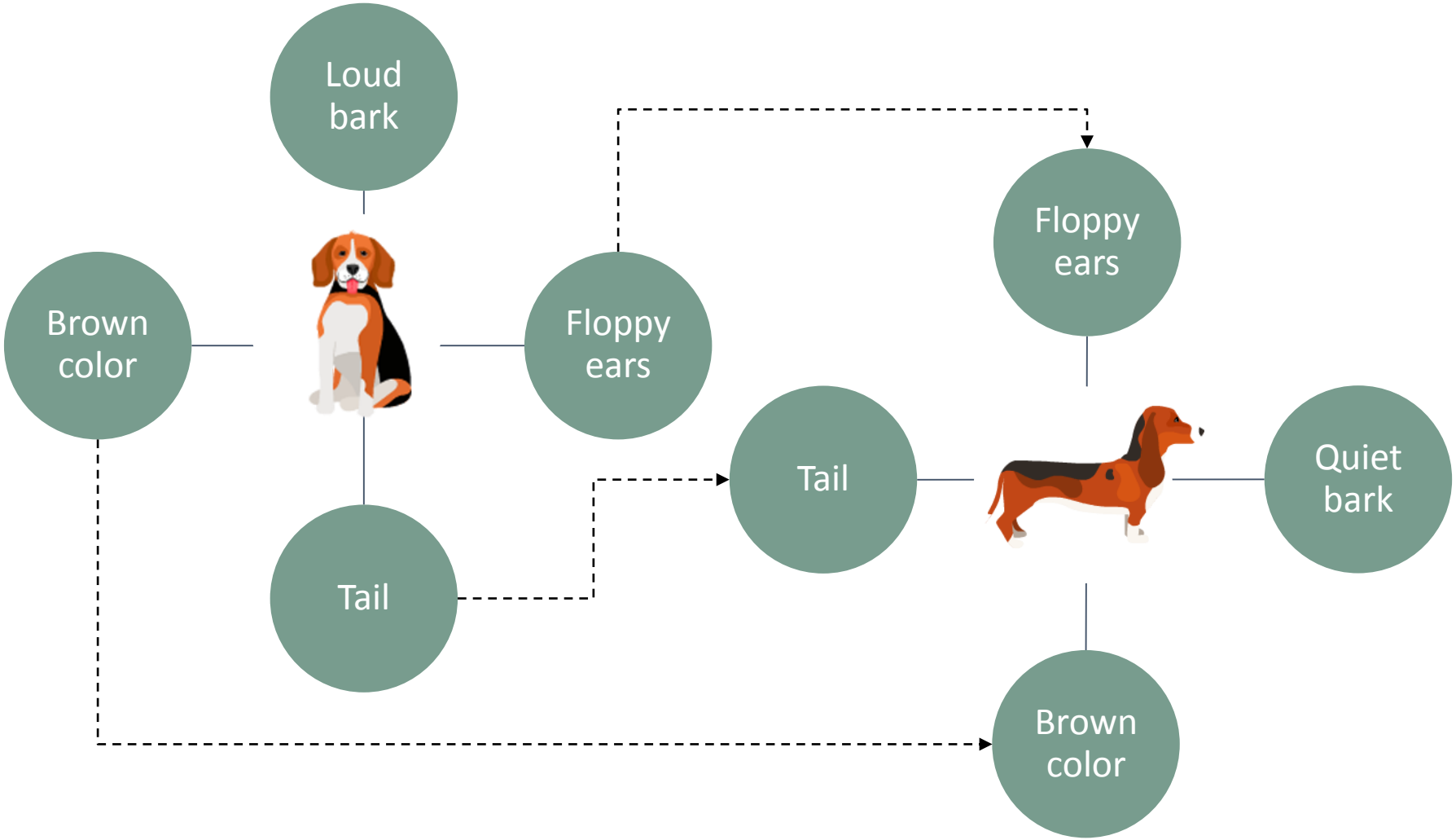




What is **encapsulation**?



What is **inheritance**?



Super class and Sub class

Brighter Blue

Super class

Class: Dog

Attributes:
- One Nose
- Two Ears
- Four Legs
- One Tail

Sub class

Class: Dachshund

Inherited Attributes:
- One Nose
- Two Ears
- Four Legs
- One Tail

Own Attributes:
- Short legs
- Long body

Class: Pug

Inherited Attributes:
- One Nose
- Two Ears
- Four Legs
- One Tail

Own Attributes:
- Short muzzle
- Compact square body

IBM

Relationships between classes

Is - A

Mammal

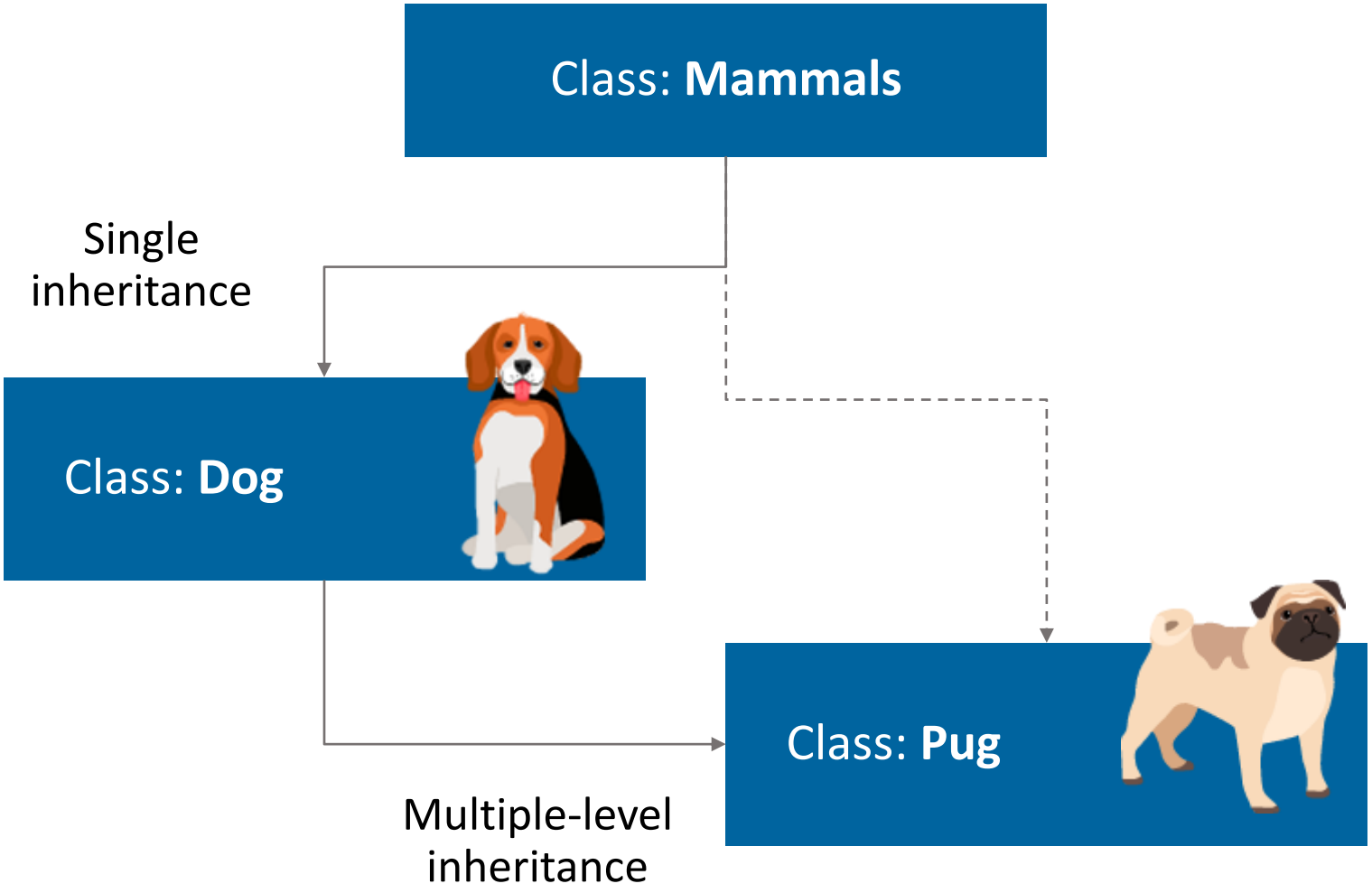
Has - A

Tail

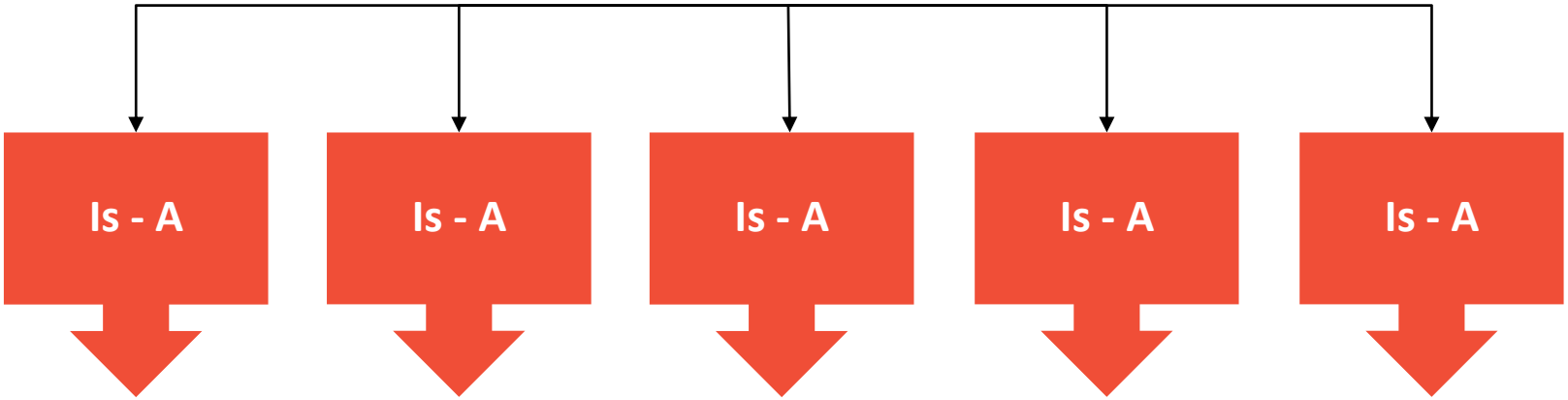
Uses - A

Legs to Run

Types of inheritance



Polymorphism



Brighter Blue

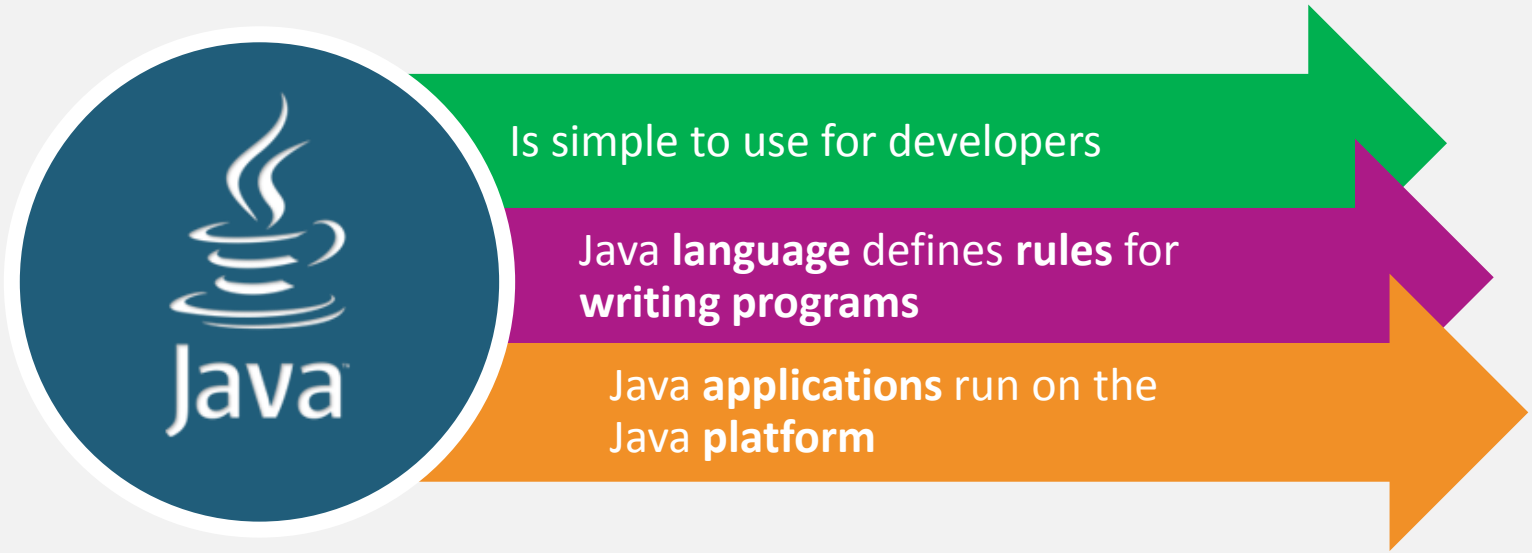
02 Getting Started with Java

IBM

What is Java?

Java is both a high-level **OOP language** and a **platform**.

With Java you can build applications that run on **multiple hardware / software platforms** without **modification**.



What are the main **Java platforms?**

Java Platform, Standard Edition 7 (Java SE 7)

Java 2 Platform, Enterprise Edition (J2EE)

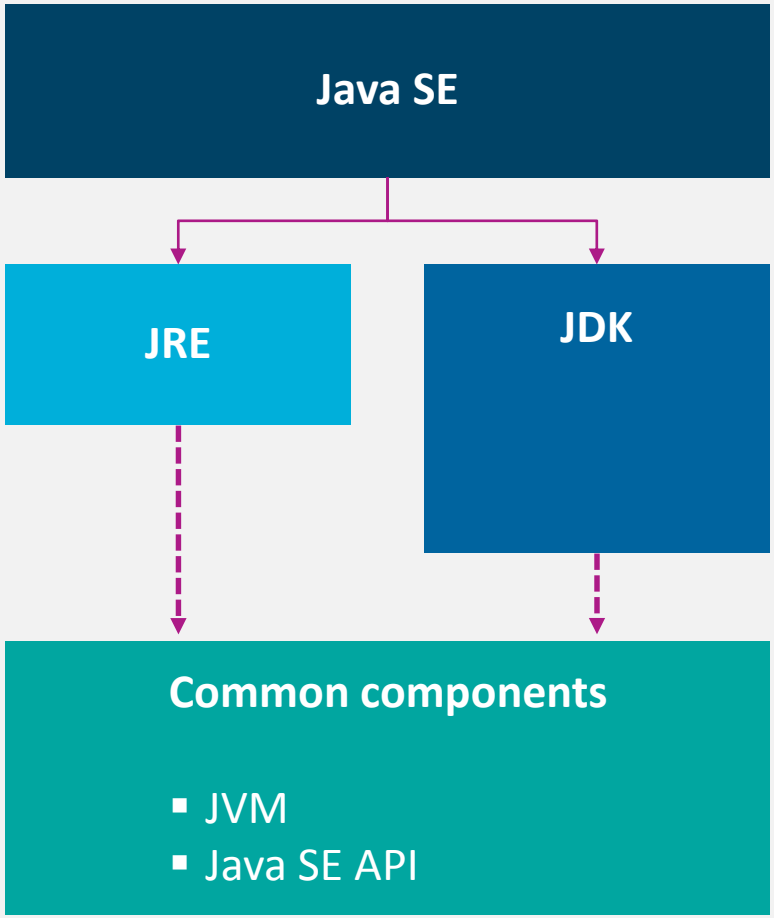
Java 2 Platform, Micro Edition (J2ME)

Elements of **Java SE**

The Java Platform is made up of two elements:

- Java SE Runtime Environment (JRE)
- Java SE Development Kit (JDK)

- Common components to both the JRE and JDK are:
  - Java Virtual Machine (JVM)
  - Java SE Application Program Interface (Java SE API)



Spot quiz

Brighter Blue

01

What are the functions of the JVM?

A

It ensures that Java is platform independent.

B

It interprets compiled Java code.

C

It groups software into libraries of related components.

D

It provides components for the creation of applets.

IBM

Spot quiz

Brighter Blue

02

Which are the components of the JRE?

A	JDK
B	JVM
C	Java compiler
D	Java SE API

IBM

Your first **Java application**

```
//this is a simple Java application

public class FirstApp {

  public static void main(String args[]) {

    System.out.println("Your first Java application");

  }
}
```

This is a simple Java application that prints out the message — **Your first Java application**.

Comments

```
//this is a simple Java application

public class FirstApp {

   public static void main(String args[]) {

      System.out.println("Your first Java application");

   }
}
```

This is a **Comment**.

You can use comments to make your code easy for other programmers to understand and use.

The class definition

```
//this is a simple Java applic

public class FirstApp {

   public static void main(String args[]) {

      System.out.println("Your first Java application");

   }
}
```

This is the **Class Definition**.

The naming convention in Java is to start class names with uppercase letters.

The main method

```
//this is a simple Java application

public class FirstApp {

   public static void main(String args[]) {

      System.out.println("Your first Java application");

   }
}
```

This is the **Main Method**.

The main method is used to start the application. Every Java application must include at least one class with a main method.

The main method's **implementation code**

```
//this is a simple Java application

public class FirstApp {

   public static void main(String args[]) {

      System.out.println("Your first Java application");

   }
}
```

This is the **Implementation Code**.

Curly brackets enclose the main method's implementation code.

Here the main method simply prints a string – "Your first Java application".

Modifiers and arguments in the main method

```
//this is a simple Java applicatio

public class FirstApp {

  public static void main(String args[]) {

                    ("Your first Java application");

}c
```

Arguments passed to the program at runtime.

Modifiers used in the Main Method.

All Java applications must have a main method with a precise signature.

Spot quiz

Brighter Blue

03

Identify the correct signature of the main method.

A

```
public static int main (String args[])
```

B

```
public static void main (String args)
```

C

```
public static void main (String args[])
```

D

```
public void main (String[])
```


IBM

Activity

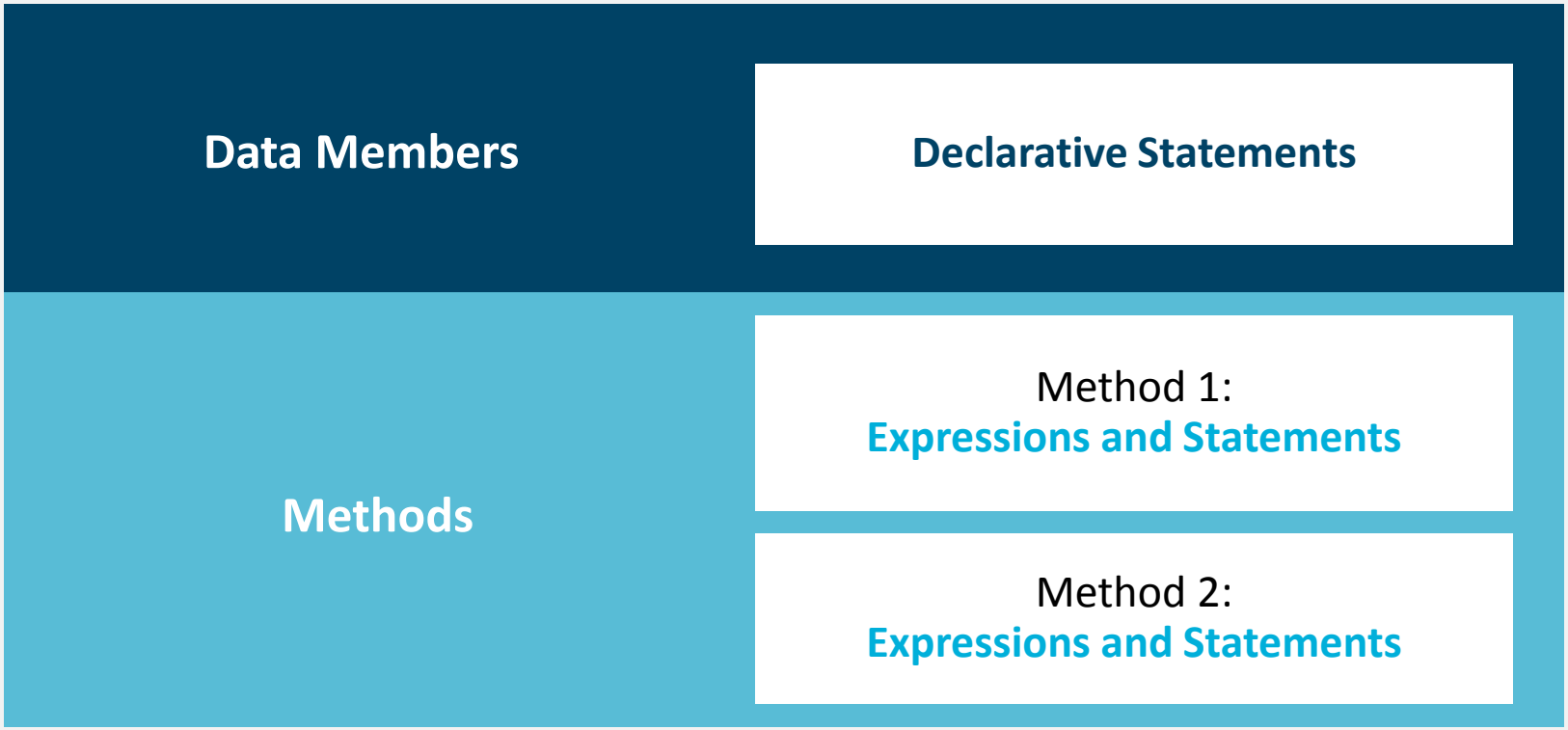
Compiling and Running a Java Application

Try out the steps to compile and run a Java application:

- Configure Eclipse for use.
- Create the FirstApp.java file
- Compile the code.
- Run the FirstApp.class file.

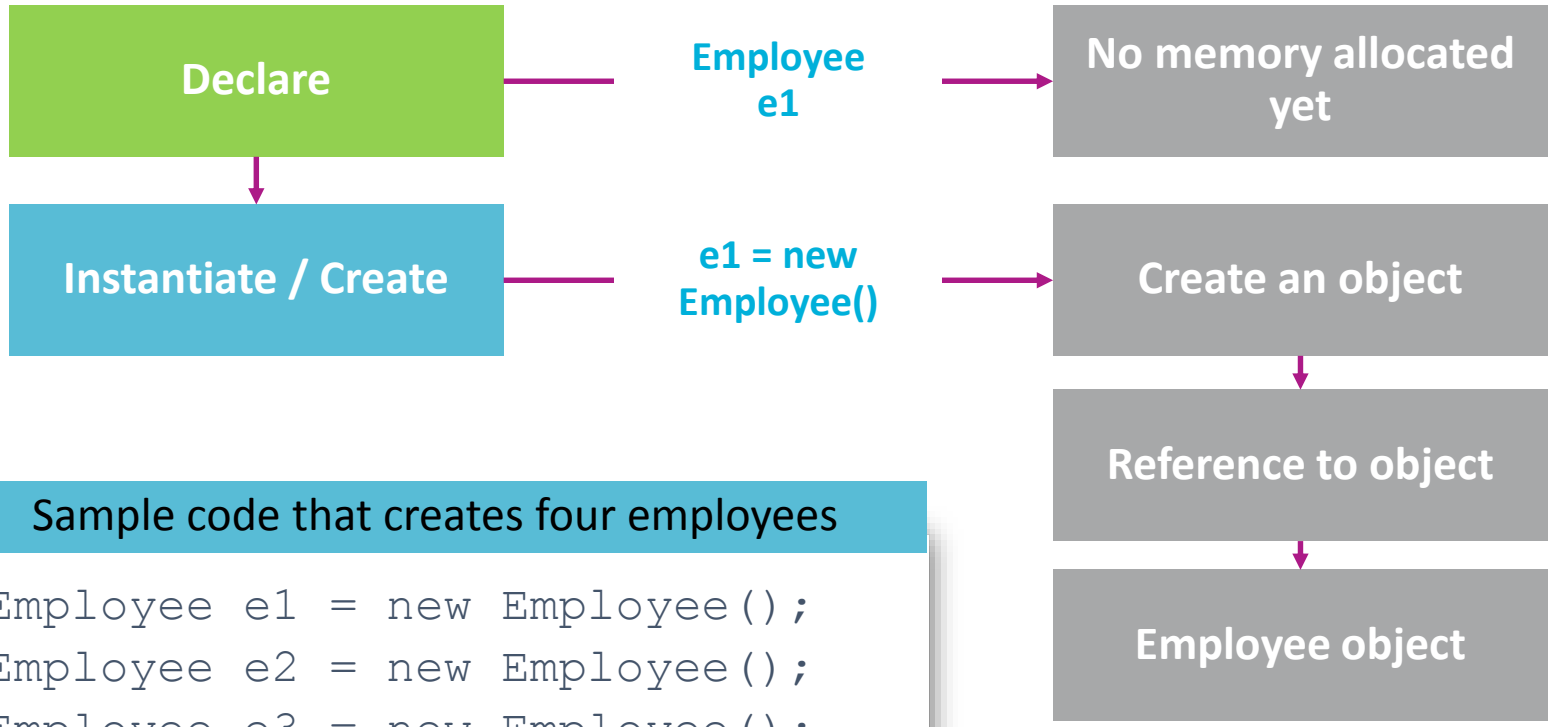
Components of a class

A Class consists of **Data members (Attributes)** and **Methods**.



Creating objects of a class

An object is an instance of a class having a unique identity.



Sample code that creates four employees

```
Employee e1 = new Employee();
Employee e2 = new Employee();
Employee e3 = new Employee();
Employee e4 = new Employee();
```

Accessing data members of a class

Assign **values** to data members of the object before using them.

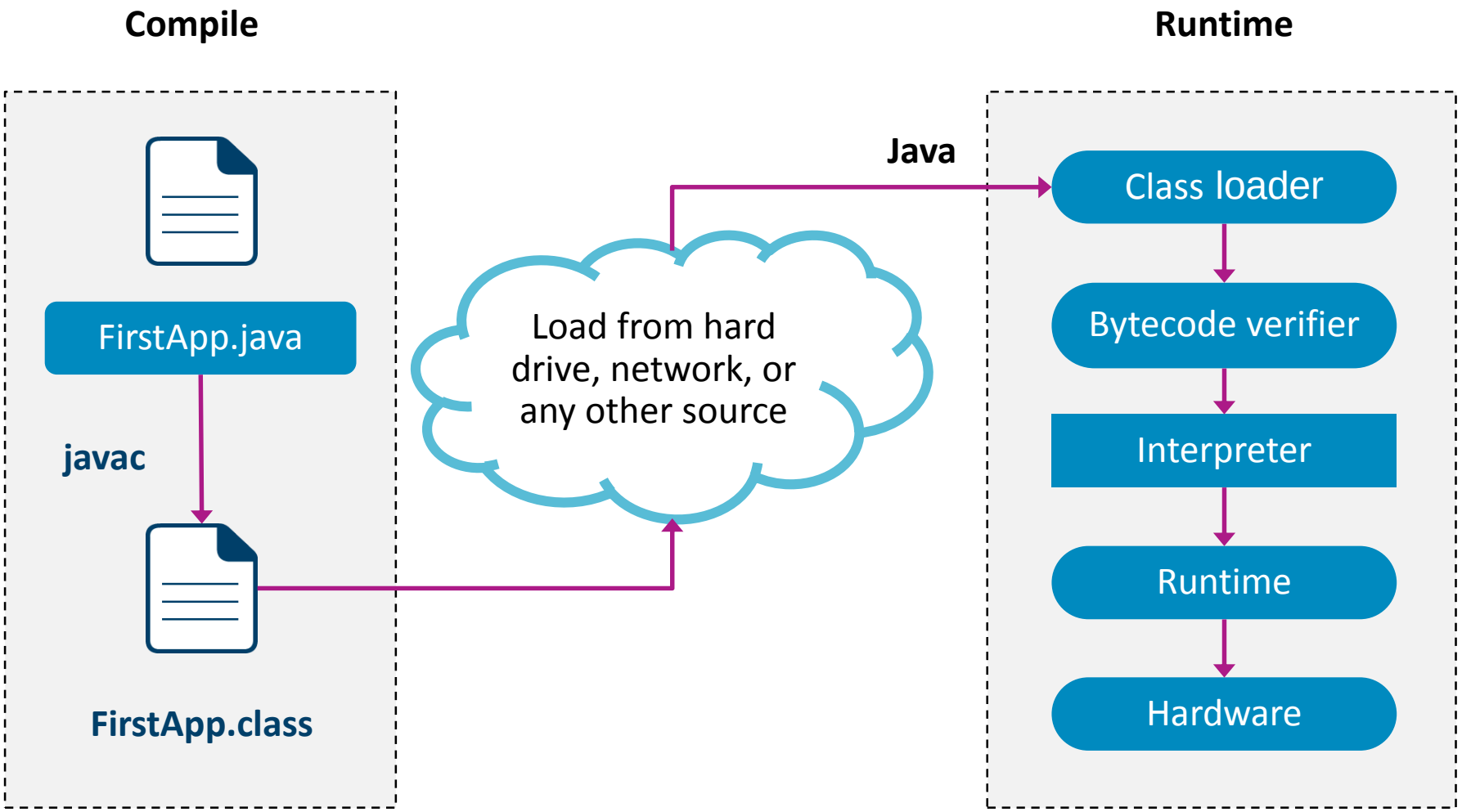
Access the data members of a class outside the class by specifying the **object name** followed by the **dot** operator and the **data member** name.

Accessing data members

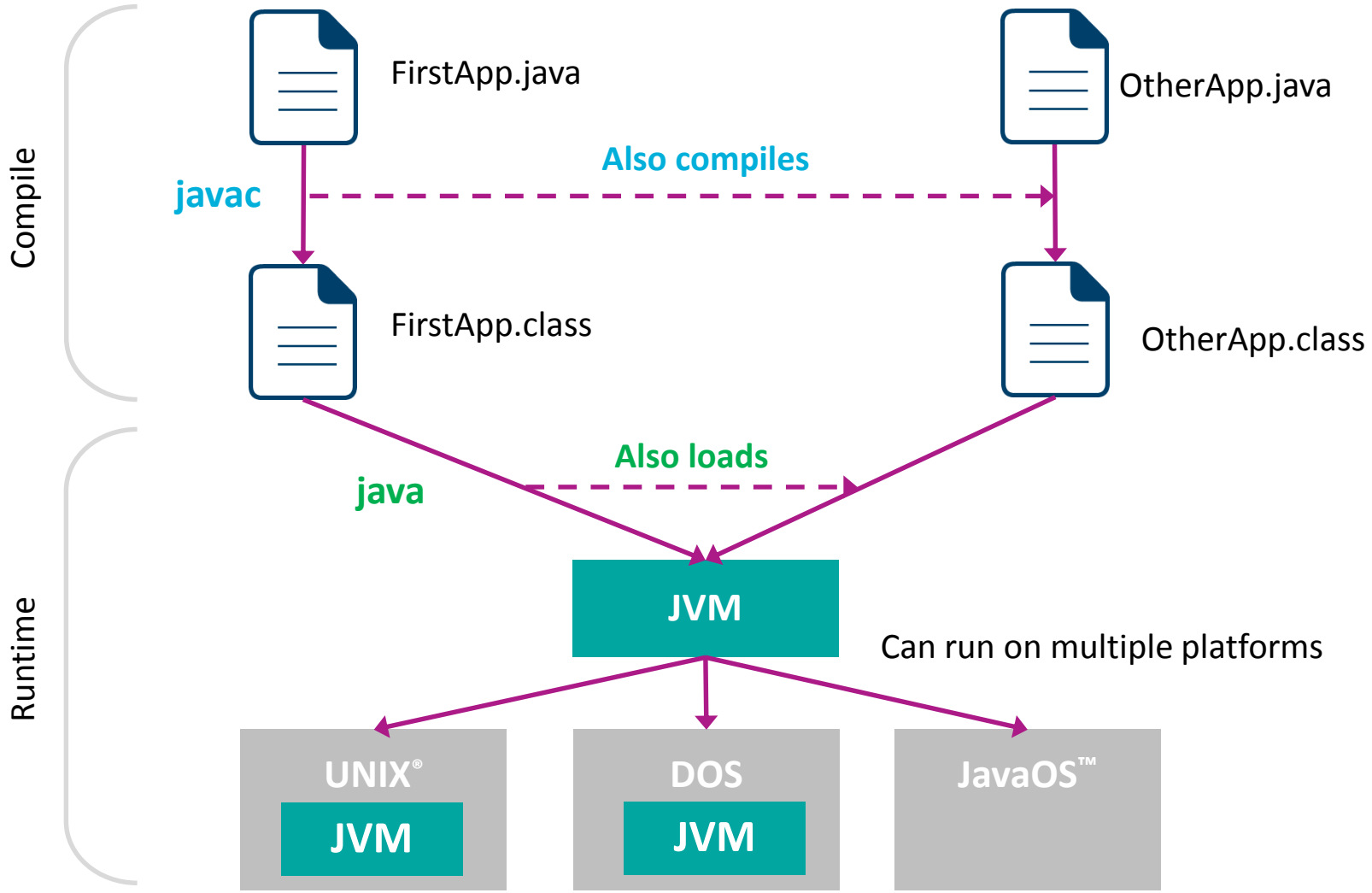
Assigning **values** to the object

```
e1.employeeName = "John";
e2.emmployeeName = "Andy";

e1.employeeID = 1;
e2.employeeID = 2;

e1.employeeDesignation = “Manager”;
e2.employeeDesignation = “Director”;
```

The Java **execution model**

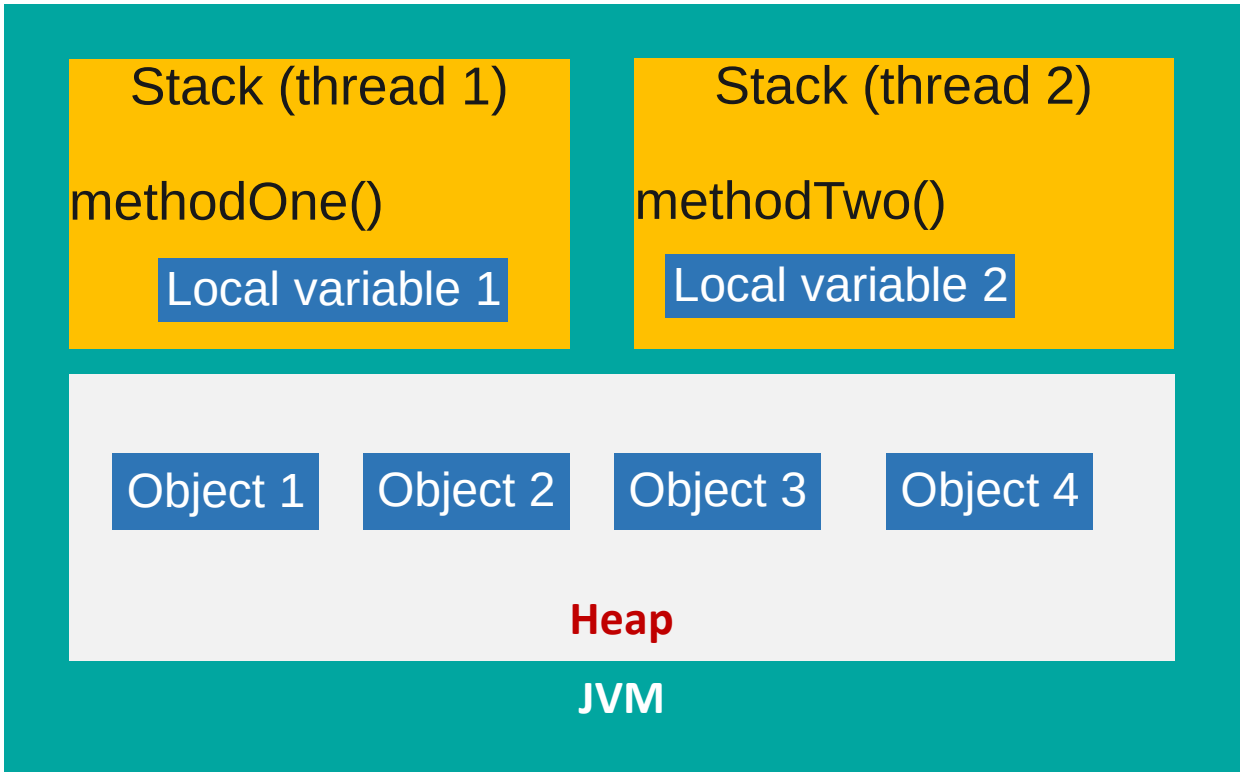


The Java **execution model** (continued)



Java **memory model**

- The Java memory model used internally in the JVM divides memory between thread stacks and the heap. This diagram illustrates the Java memory model from a logic perspective:



Spot quiz

Brighter Blue

04

Suppose you are required to compile and run the code for a class named Calculator. Select the commands to do this.

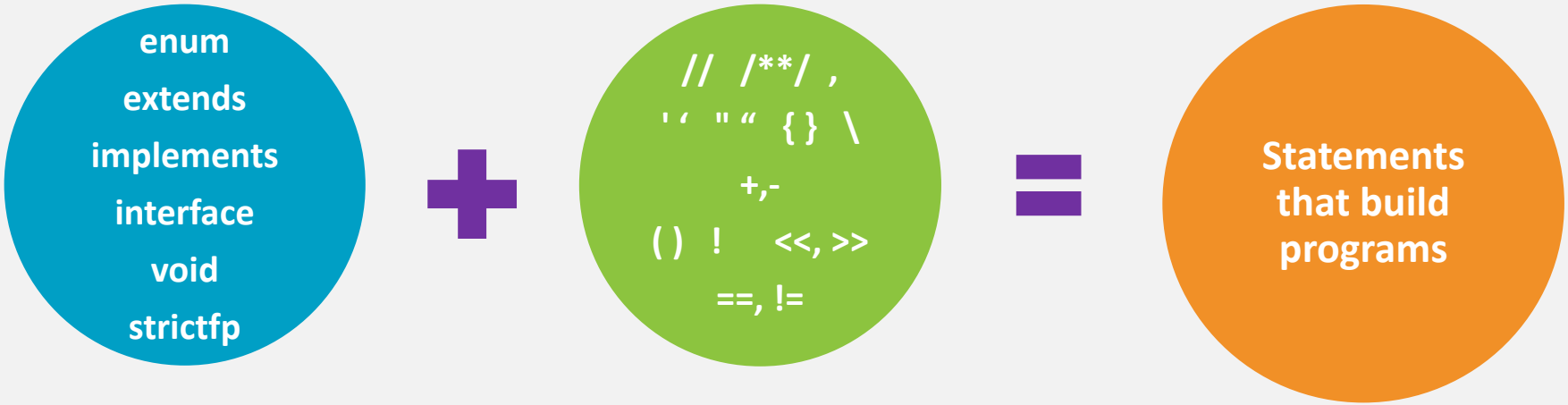
A	java Calculator
B	java Calculator.class
C	javac Calculator
D	javac Calculator.java

Brighter Blue

03 Basic Lexical Elements in Java

IBM

What are **lexical elements**?



Lexical elements are the words and symbols that make up a programming language.

Statements and statement blocks

```
/**
* This class contains basic employee info
*/
public class Employee {
    String name;
    String role;
    int phoneNumber;
    double salary;

    public static void main(String args[]) {
        Employee emp = new Employee("Jon Woo","Developer");
        //create an Employee object
        emp.printEmpInfo();
        //print the Employee data
}
/* this method prints out all the employee's
* details to the console
*/
```

Statement

Statement Block

Comments

```
/**
* This class contains basic employee info
*/
public class Employee {
    String name;
    String role;
    int phoneNumber;
    double salary;

    public static void main(String args[]) {
        Employee emp = new Employee("Jon Woo","Developer");
        //create an Employee object
        emp.printEmpInfo();
        //print the Employee data
}
/* this method prints out all the employee's
* details to the console
*/
```

Doc comment

In-line comment

multi-line comment

Spot quiz

Brighter Blue

01

Suppose you want to include some comments as part of the documentation of a piece of code. Which notation do you use to start this type of comment?

A

```
// my comment…
```

B

```
/* my comment…
```

C

```
/** my comment…
```


IBM

Java identifiers

```
/**
* This class contains basic employee info
*/
public class Employee {
    String name;
    String role;
    int phoneNumber;
    double salary;

    public static void main(String args[]) {
        Employee emp = new Employee("Jon Woo","Developer");
        //create an Employee object
        emp.printEmpInfo();
        //print the Employee data
}
/* this method prints out all the employee's
* details to the console
*/
```

Employee, name, role, phoneNumber, salary are all **identifiers**

Naming identifiers

While naming your identifiers always remember that:

Identifiers are **case sensitive**

Only certain characters can start an identifier

Identifiers can consist of any number of characters

Spot Quiz

Brighter Blue

02

Identify valid identifiers for use in code.

A	`$role`
B	`&_printEmployee$`
C	`_salary`
D	`phoneNumber`

IBM

What are **Java keywords**?

Keywords are reserved words in Java. Each keyword has a specific meaning that remains the same from one program to another – you can not use a keyword as an identifier.

Java keywords are grouped into the following categories:

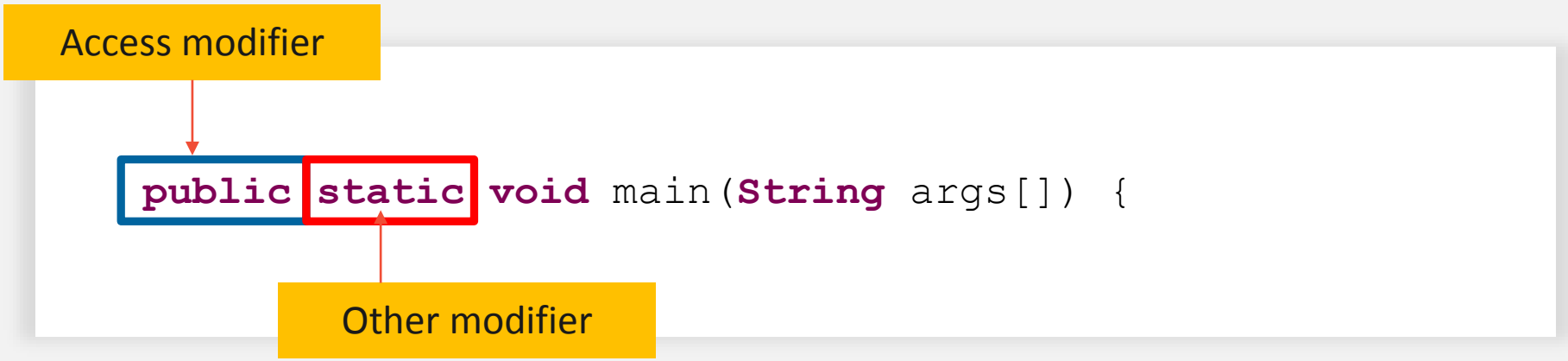
- Access modifiers
- Other modifiers
- Flow control
- Exceptions
- Class and interface declarations
- Primitive types
- Namespace keywords
- Object related keywords

Access modifiers and Other modifiers

Brighter Blue

Access modifiers include the keywords **private**, **protected**, and **public**.

Other modifiers include **abstract**, **final**, **native**, **static**, **synchronized**, **transient**, and **volatile**.



The keyword **public** declares a piece of code totally accessible to the rest of a program, whereas private and protected restrict access.

Default access modifier means we **do not explicitly declare** an access modifier **(i.e. no keyword)** for a class, field, method, etc. A variable or method declared without any access control modifier is available to any other class in the same package..

Private, protected, and public access modifiers: examples

Example of **Private Access Modifier**:

```
public class Logger {
   private String format;
   public String getFormat() {
      return this.format;
   }
   public void setFormat(String format) {
      this.format = format;
   }
}
```

Example of **Protected Access Modifier**:

```
class AudioPlayer {
   protected boolean openSpeaker(Speaker sp) {
      // implementation details
   }
}

class StreamingAudioPlayer {
   boolean openSpeaker(Speaker sp) {
      // implementation details
   }
}
```

Example of **Public Access Modifier**:

```
public static void main(String[] arguments) {
   // ...
}
```

Flow control

The keywords that control the sequence of execution of the statements in a Java program are **break**, **case**, **continue**, **default**, **do**, **else**, **for**, **if**, **return**, **switch**, **while** and **assert**.

The code below that shows how different statements are executed depending on the value of a variable, using the **if** and **else** keywords.

```
if (a==b) {
   b = c+d;
   }

else {
   a += 1;
}
```

Exceptions-related modifiers

The exception-related modifiers are **catch**, **finally**, **throw**, **throws**, and **try**.

You use exceptions to identify and report problems that may occur while a program is executing.

An example of using the try and catch keywords:

```
try {
    readFromFile("myfile.txt");
    //other risky code
    }

catch (Exception e) {
     System.out.println("Error!");
     //other error handling code
  }
```

Class and Interface declarations

Keywords that you use to declare classes and interfaces are **class**, **enum**, **extends**, **implements**, **interface**, **void**, and **strictfp**.

You use the **interface** keyword when you are defining an interface, just as you use the **class** keyword when you are defining a class as shown below.

```
public class Employee implements EmployeeInfo {

    //class declaration comes here

}
```

Namespace keywords

The namespace keywords are **import** and **package**.

The **import** keyword provides a shorthand way of referring to other classes in the current class.

```
import java.io*;
```

The **package** keyword is used to group related class files in a directory.

```
package Employees;
```

Spot quiz

Brighter Blue

03

Identify the Java keywords you use to modify access.

A	package
B	private
C	protected
D	public

IBM

Spot quiz

Brighter Blue

04

Which one of these lists contains only Java programming language keywords?

A	class, if, void, long, Int, continue
B	instanceof, native, finally, default, throws
C	try, virtual, throw, final, volatile, transient
D	strictfp, constant, super, implements, do
E	byte, break, assert, switch, include

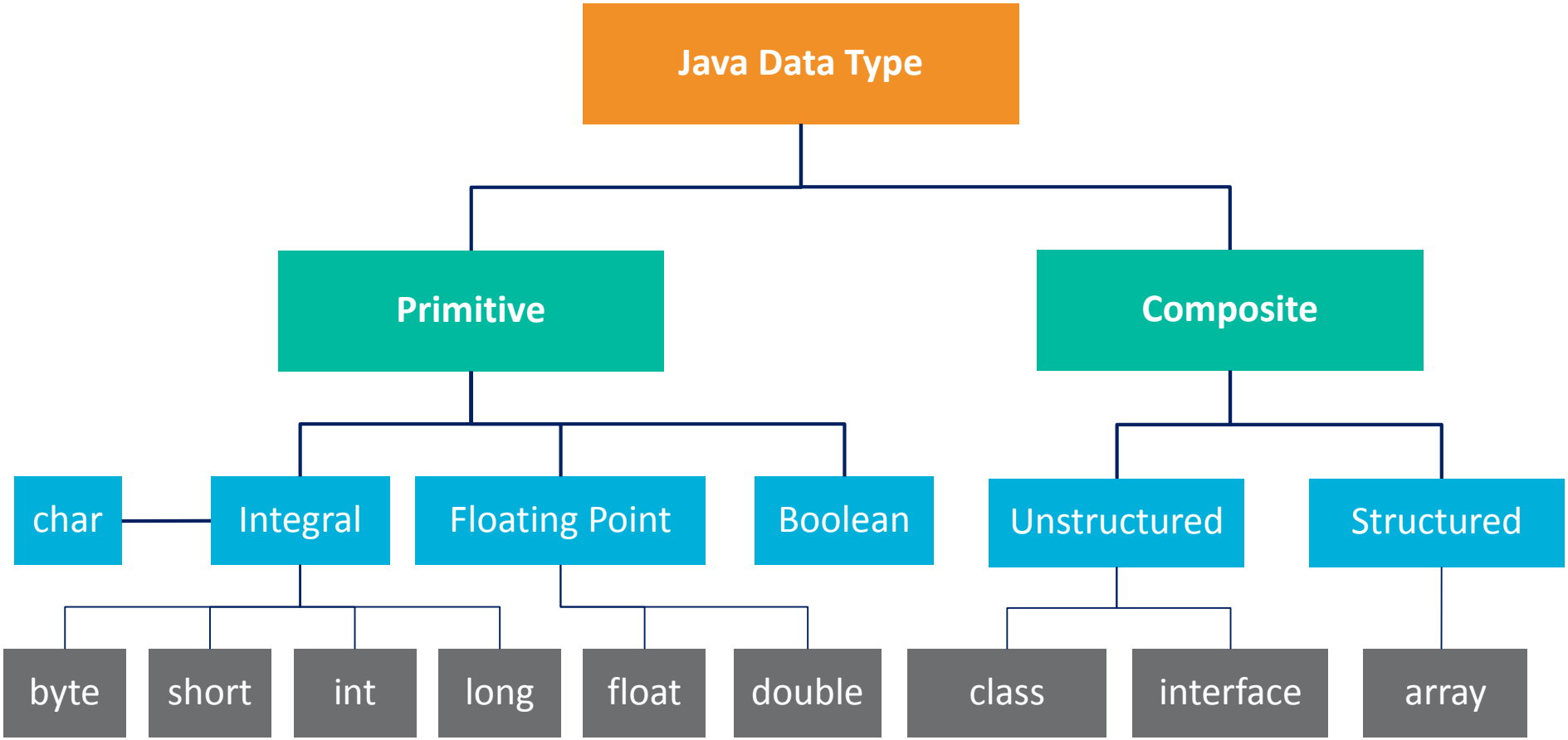
IBM

Brighter Blue

04 Java Primitive Data Types

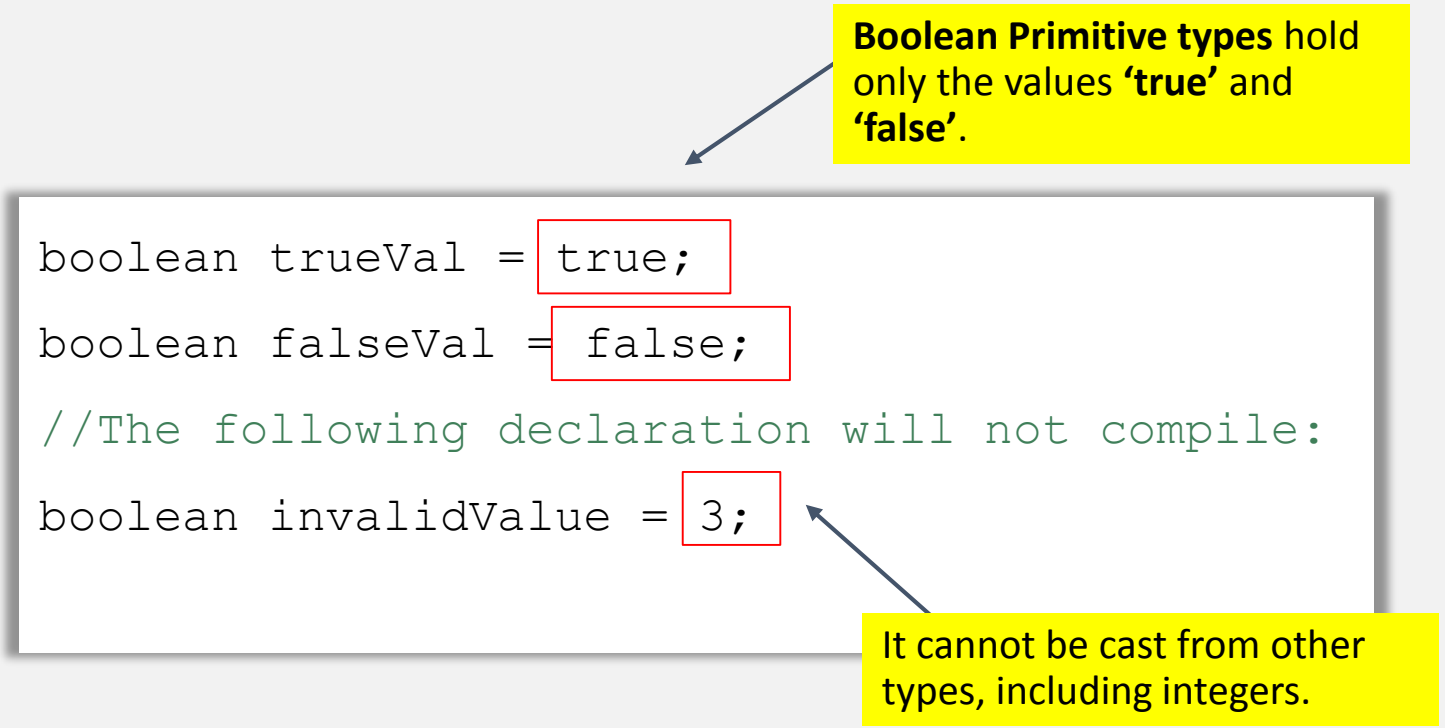
IBM

Java Primitive data types

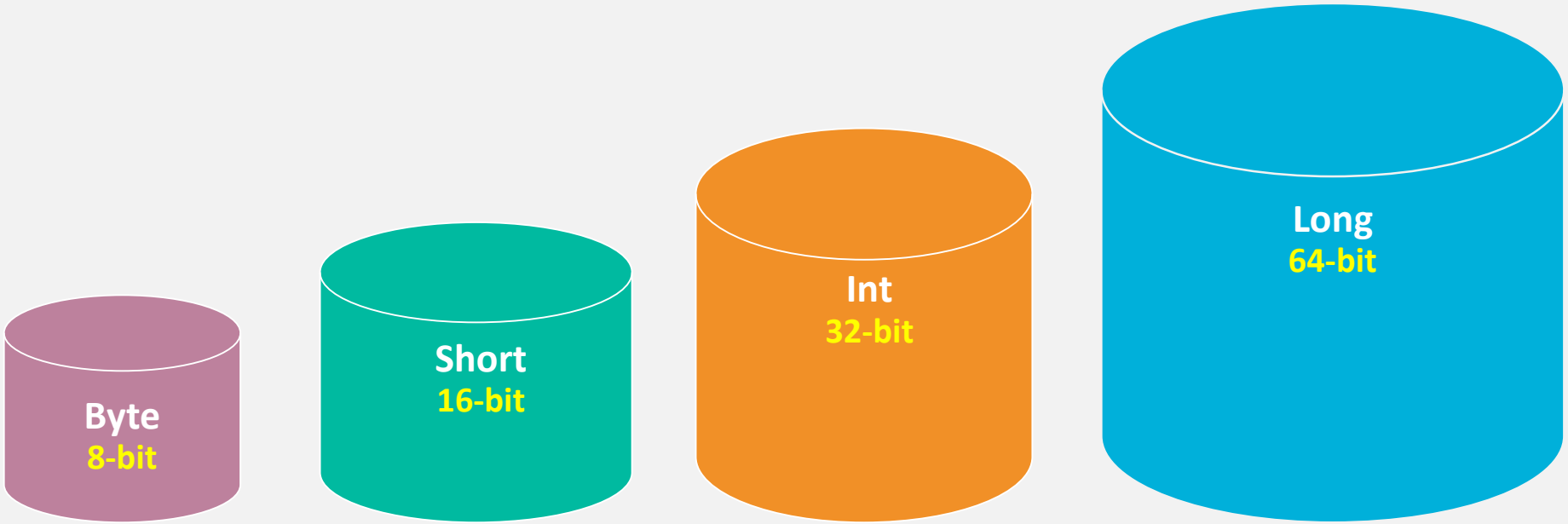


Boolean Primitive type

You can declare boolean Primitive types as shown here:



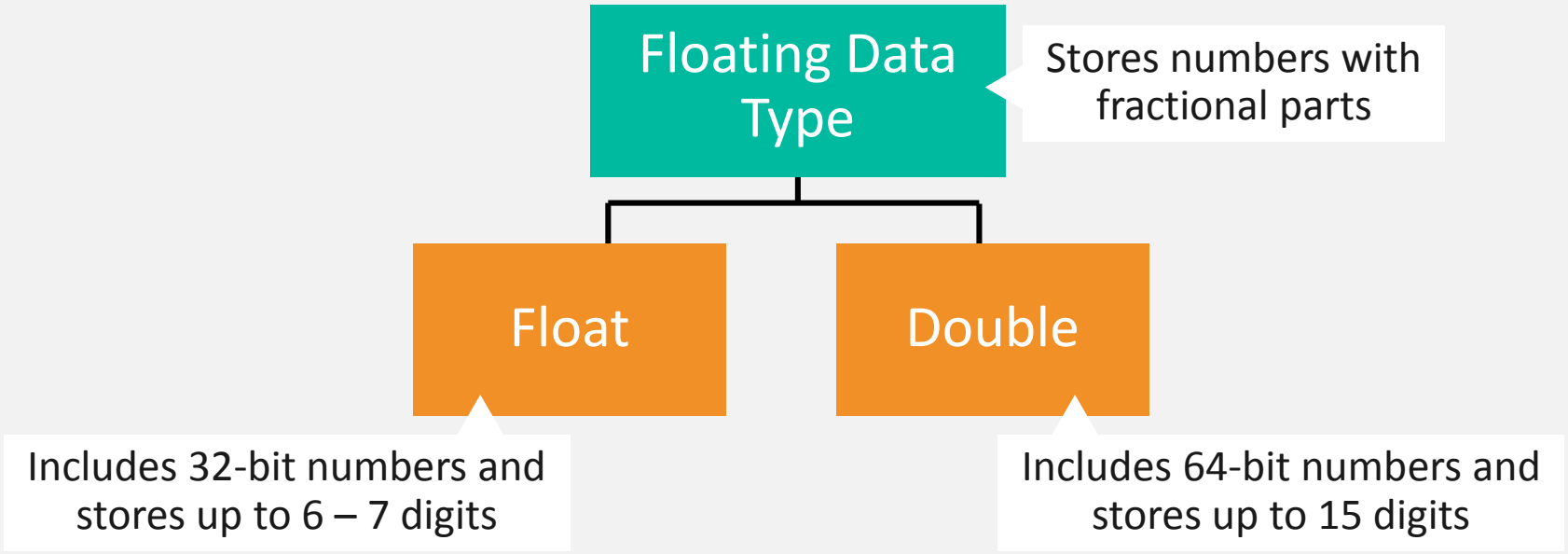
Integral data type



The variables of integral types are declared as :

- **byte myByte = 2;**
- **short myShort = 2;**
- **int myInt = 2;**
- **long myLong = 2L;**

Floating-point data type



Floating-point data type (continued)

Brighter Blue

You can declare the floating-point types as follows:

```
float myFloat = 23.5F;
double myDouble = 2000000.55;
```

Spot Quiz

Brighter Blue

01

Which of the following are floating data types?

A Int

B Float

C Long

D Double

IBM

Spot Quiz

02

What is the **output** of this program?

```
1.    class CalculateArea {
2.        public static void main(String args[])
3.        {
4.              double r, pi, a;
5.              r = 5.8;
6.              pi = 3.14;
7.              a = pi * r * r;
8.              System.out.println(a);
9.        }
10.    }
```

A 105.6296

B 105

C 105.62

D 105.62960000

Rules for **using variables**

- You declare a variable by preceding the name of the variable with its type.

```
datatype variableName;

    int amount;
    boolean result;
```

Examples of variable declaration

- You should always provide an initial value for local variables or the code will generate a compiler error.

```
datatype variableName = value;

    short amount = 560;
```

Example of short variable declaration

Types of **variables**

Local variables: Local variables are declared in methods, constructors, or blocks.

Instance variables: Instance variables hold values that must be referenced by more than one method, constructor or block so instance variables are declared in a class, but outside a method, constructor or any block.

Static variables: Static variables (or Class variables) are declared with the **static** keyword in a class, but outside a method, constructor or a block.

There would only be **one copy** of each **static variable per class**, regardless of how many objects are created from it.

```
int counting(int n1,int n2)
  {
  //local variable example
  int count;
  count = n1 + n2;
  return(count);
  }
```

```
Class Person
{
//Instance variables examples
String name;
int    age;
String gender;

//Static variable example
static String city = "New York";

};

Person P1,P2,P3;
```

Spot Quiz

Brighter Blue

03

Suppose you want to write a program that monitors your expenses at the end of each month. Which variable declaration do you use to describe your annual income of one million dollars?

A

```
double annualSalary
= 1000000.00;
```

B

```
float annualSalary =
1000000.00;
```

C

```
long annualSalary =
1000000.00;
```


IBM

Spot Quiz

Brighter Blue

04

Which three are valid declarations of a char?

```
1. char c1 = 064770;
2. char c2 = 'face';
3. char c3 = 0xbeef;
```

```
4. char c4 = \u0022;
5. char c5 = '\iface';
6. char c6 = '\uface';
```

A. 1, 2, and 4

B. 1, 3, and 6

C. 3 and 5

D. 5 only

IBM

Spot Quiz

Brighter Blue

05

Which one is a valid declaration of a boolean?

A

```
boolean b1 = 0;
```

B

```
boolean b2 =
'false';
```

C

```
boolean b3 = false;
```

D

```
boolean b4 =
Boolean.false();
```

E

```
boolean b5 = no;
```


IBM

Spot Quiz

Brighter Blue

06

Which three are valid declaration of a float?

A

```
float f1 = -343;
```

B

```
float f2 = 3.14;
```

C

```
float f3 = 0x12345;
```

D

```
float f4 = 42e7;
```

E

```
float f5 = 2001.0D;
```

F

```
float f6 = 2.81F;
```


IBM

What is a **literal**?

In the following declaration, the 4000 is a numeric literal that represents the number four thousand.

```
int amount = 4000;
```

What is a **string**?

String literals are defined between **double quotes**.

More on this later.

```
String myString = "String literal";
```

Example of a string

Integral literals

Brighter Blue

Code examples of integer literals:

```
//hex. preceded by 0x
byte myByte = 0xA6;
long myLong = 0xBD6700F;

//octals preceded by '0'
short myShort = 01534;
int myInt =     0123313;

//A 32-bit integer literal
int myInt = 0xAB153FE6

//A 64-bit integer literal
long myLong =  76354L;
long myLong2 = 63521;
```

Hexdigits can be uppercase or lowercase.

Octals precede the literals with zero.

To specify that an integer literal is a long integer with a 64-bit value, you place an uppercase or lowercase "L" after it.

Char literals

Char literals are enclosed in single quotes:

```
char d = 'd';
char space = '\u0020'

// this is the space character
```

Floating-point literals

- You indicate floating-point literals by appending an uppercase or lowercase "F" to the end of a literal name.
- D or d indicates that a literal is of type double.

```
double myDouble = 12.4D;
float myFloat = 12.4F;
double mySecondDouble = 12.4;
float mySecondFloat = (float) 12.4;
double myExponential = 5e3;
```

Code example of floating-point literals

Boolean literals

Valid boolean data type values are the reserved literals true and false.

```
boolean trueValue = true;
boolean falseValue = false;
```

Code examples using boolean literals

The other types of **literals**

Brighter Blue

The other 3 types of literals are:

Char literals - enclosed in single quotes:

```
char d = 'd';
char space = '\u0020"

// this is the space character
```

Floating point literals – indicated by appending an uppercase or lowercase **F** to the end of the literal name:

```
double myDouble = 12.4D;
float myFloat = 12.4F;
double mySecondDouble = 12.4;
float mySecondFloat = (float)12.4;
double myExponential = 5e3;
```

Boolean literals –Valid boolean data type values are the reserved literals **true** and **false**:

```
boolean trueValue = true;
boolean falseValue = false;
```

Spot Quiz

Brighter Blue

07

How would you specify that an integer literal is a long integer with a 64 bit value?

A	By placing an upper or lowercase **L** after it
B	By preceding the literals with 0 (zero)
C	By appending an upper or lowercase **F** after it
C	By indicating an upper or lowercase **D**

IBM

Spot quiz

Brighter Blue

08

Which type of literal is written in a pair of single quotes?

A Integer

B Character

C Boolean

D Float

IBM

Brighter Blue

05 Strings and Arrays in Java

IBM

What is a **string**?

A **string** is an **object** that **represents a sequence of characters.** A string class is used to create string object, and **can store multiple objects.**

There are two ways to create a string object: by **string literal**, and by the **new operator**.

- Java string literal is created by **using double quotes**:

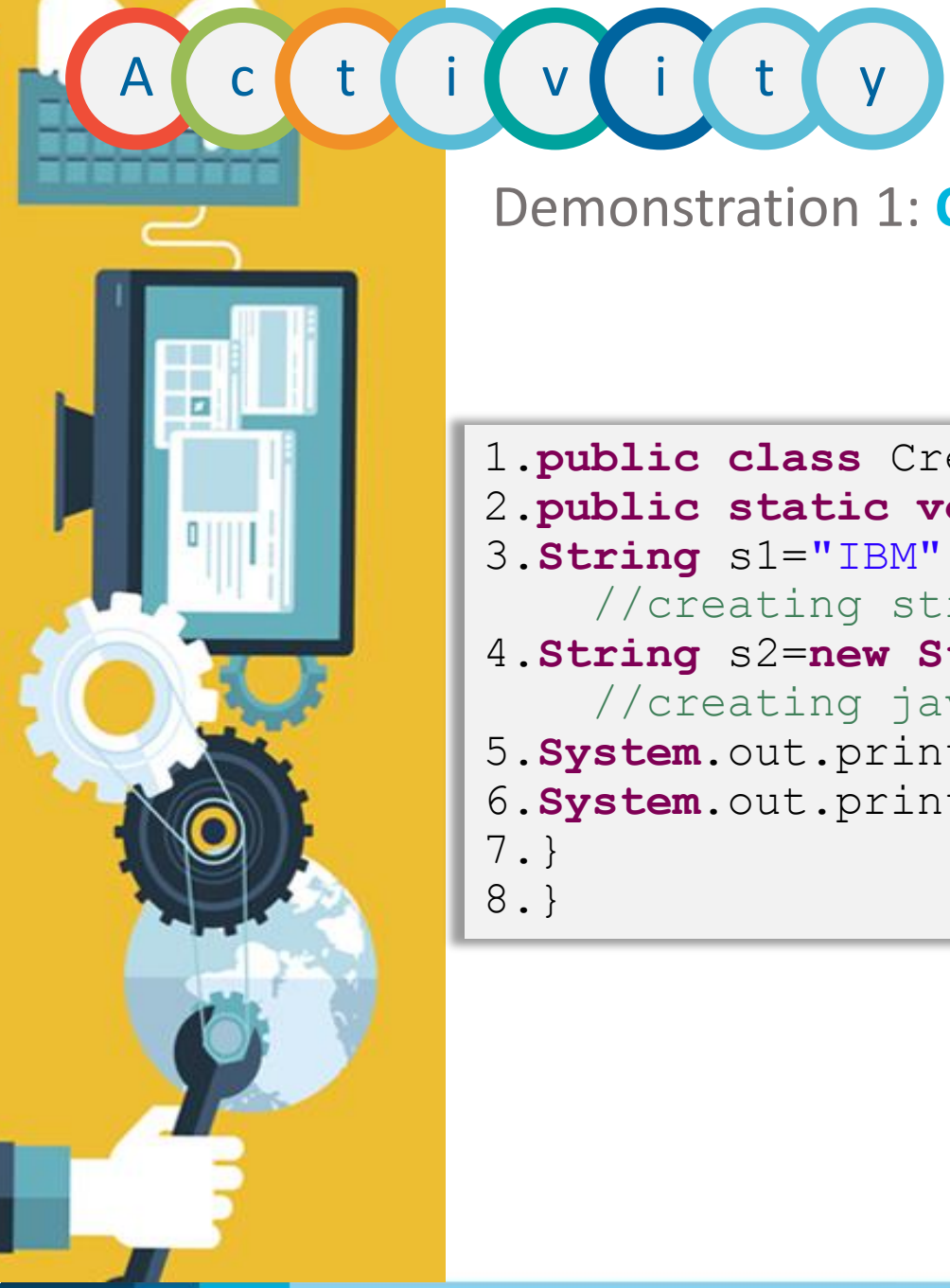
```
String greeting = "Hello!";
```

- You can create a string object using the **new operator.**

```
String myString = new String(" A String");
```

Each time you create a string using **new**, a new **string** object is created, even if you use the **same literal**.

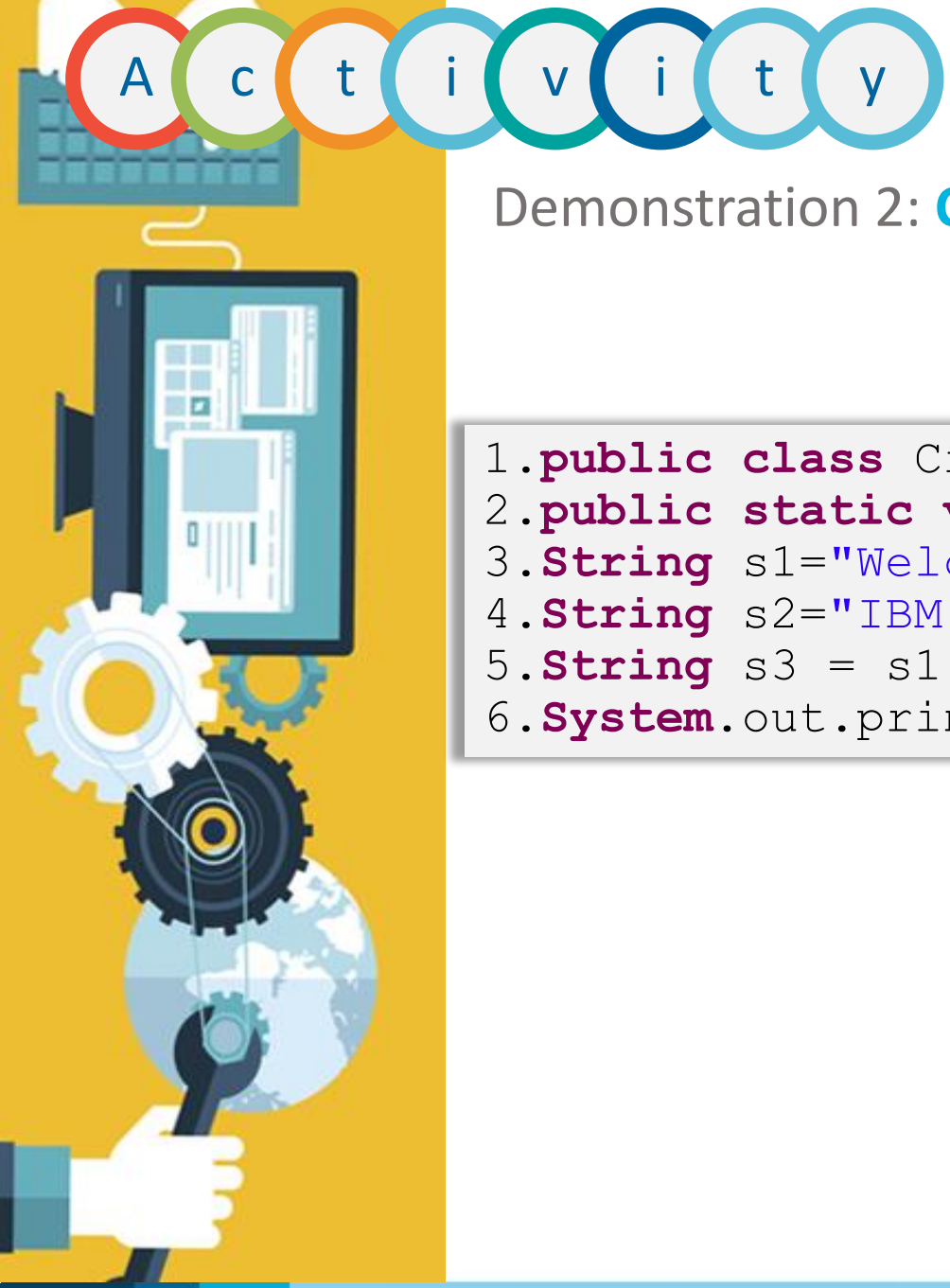
```
String myFirstString = new String (" A String literal");
      // A new string is created
String mySecondString = new String (" A String literal");
```



Brighter Blue

Demonstration 1: Creation of Strings

```
1.public class CreatingString{
2.public static void main(String args[]){
3.String s1="IBM";
    //creating string by java string literal
4.String s2=new String("IBM India");
    //creating java string by new keyword
5.System.out.println(s1);
6.System.out.println(s2);
7.}
8.}
```

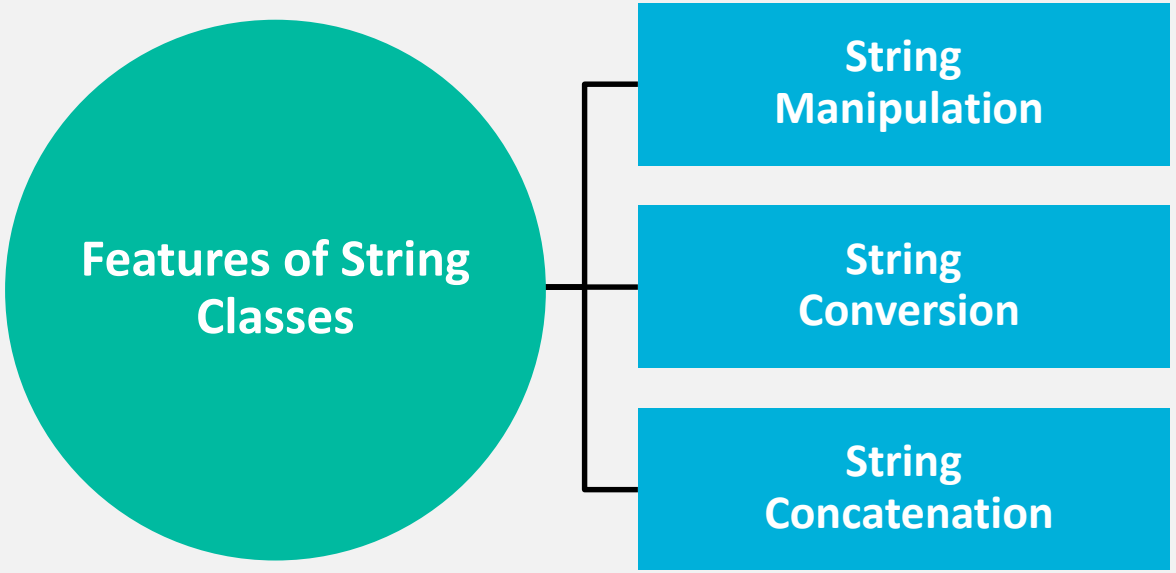


Brighter Blue

Demonstration 2: **Creation of Strings**

```
1.public class CreatingString{
2.public static void main(String args[]){
3.String s1="Welcome to Java Programming";
4.String s2="IBM India";
5.String s3 = s1 + s2;
6.System.out.println("Final String:" +s3);
```

Features of **string classes**



String **manipulation**

Brighter Blue

Strings in Java are implemented as **instances** of the String class, and as a result, you can use a range of **Java methods to manipulate** them.

```
String start = "Welcome to Java";
     start.toUpperCase();
```

Consider the code that uses the **toUpperCase** method to convert the string, start, to uppercase.

String **conversion**

Consider the code that uses the **valueOf** method to return the string representations of an integer and a double, respectively.

```
String intVal = String.valueOf(10000);
String doubleVal = String.valueOf(10.2/3.4);
```

String **concatenation**

- The following code concatenates two strings and then outputs the string "This is a string definition".

```
String myString = "This is a " + "string definition";
```

- In the following example, a string is concatenated with a number:

```
int x = 10;
     String numApples = "I have "+ x + " apples";
```

Spot Quiz

Brighter Blue

01

Which of the following are valid declarations of a String?

A	String s1 = null;
B	String s2 = 'null';
C	String s3 = (String) 'abc';
D	String s4 = (String) '\ufeed';

IBM

StringBuffer class

Brighter Blue

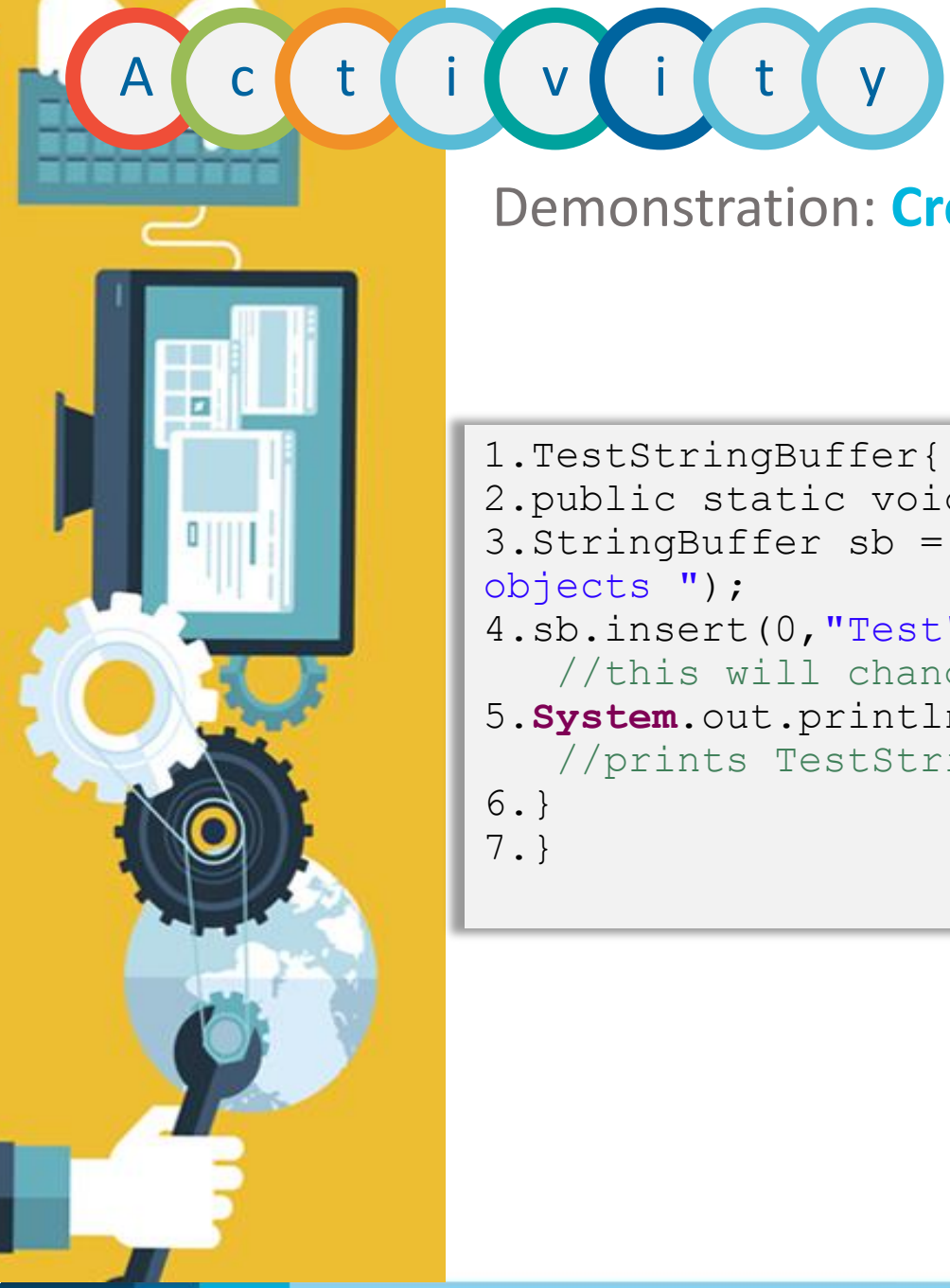
The java.lang.StringBuffer class is a thread safe, mutable sequence of characters. It is similar to a string class but can be modified.

You create a StringBuffer object using the **new** keyword

```
StringBuffer sbuf = new StringBuffer("Hello");
```

String vs. StringBuffer

String	StringBuffer
String class is immutable.	StringBuffer class is mutable.
String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you cancat strings.
String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.



Brighter Blue

Demonstration: **Creation of string buffer**

```
1.TestStringBuffer{
2.public static void main(String args[]){
3.StringBuffer sb = new StringBuffer("StringBuffer
objects ");
4.sb.insert(0,"Test");
   //this will change the string
5.System.out.println(sb);
   //prints TestStringBuffer objects
6.}
7.}
```

StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized.

You create a StringBuilder object using the **new** keyword

```
StringBuilder sb = new StringBuilder("Hello");
```

StringBuffer vs. StringBuilder

StringBuffer	StringBuilder
StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.

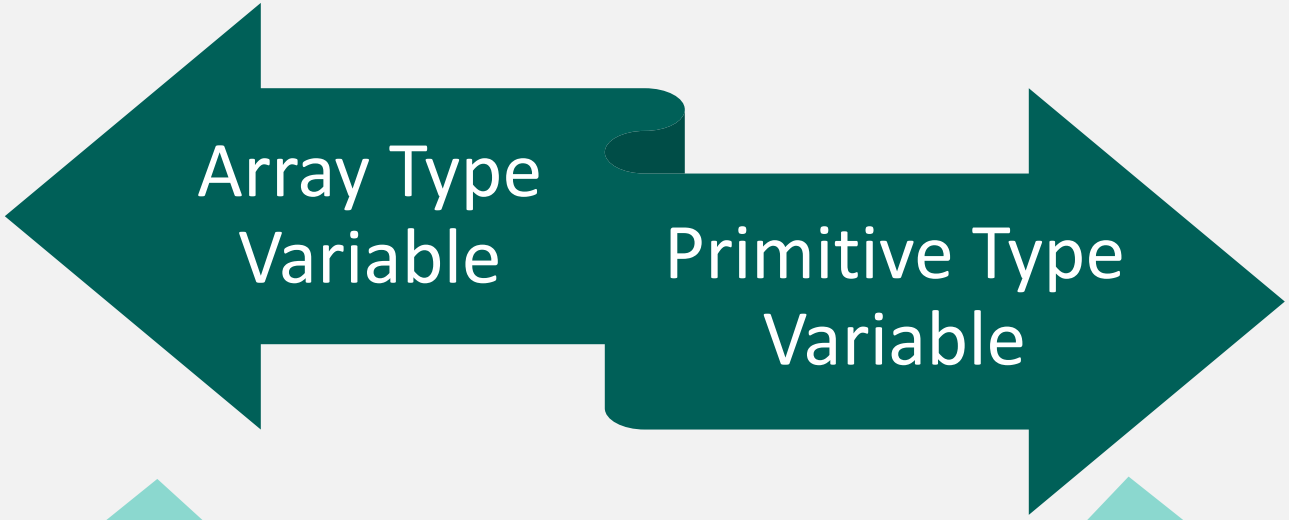
Immutable classes

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable.

We can also create immutable class by creating final class that have final data members as the example given below:

```
public final class Employee{
final String idcardNumber;

   public Employee(String idcardNumber){
   this.idcardNumber=idcardNumber;
   }

   public String getidcardNumber(){
   return idcardNumber;
   }

}
```

Difference between **array** and **primitive variables**



Spot Quiz

Brighter Blue

02

Identify a situation in which you need to use an array.

A

You need to create a string that can be modified at a later stage.

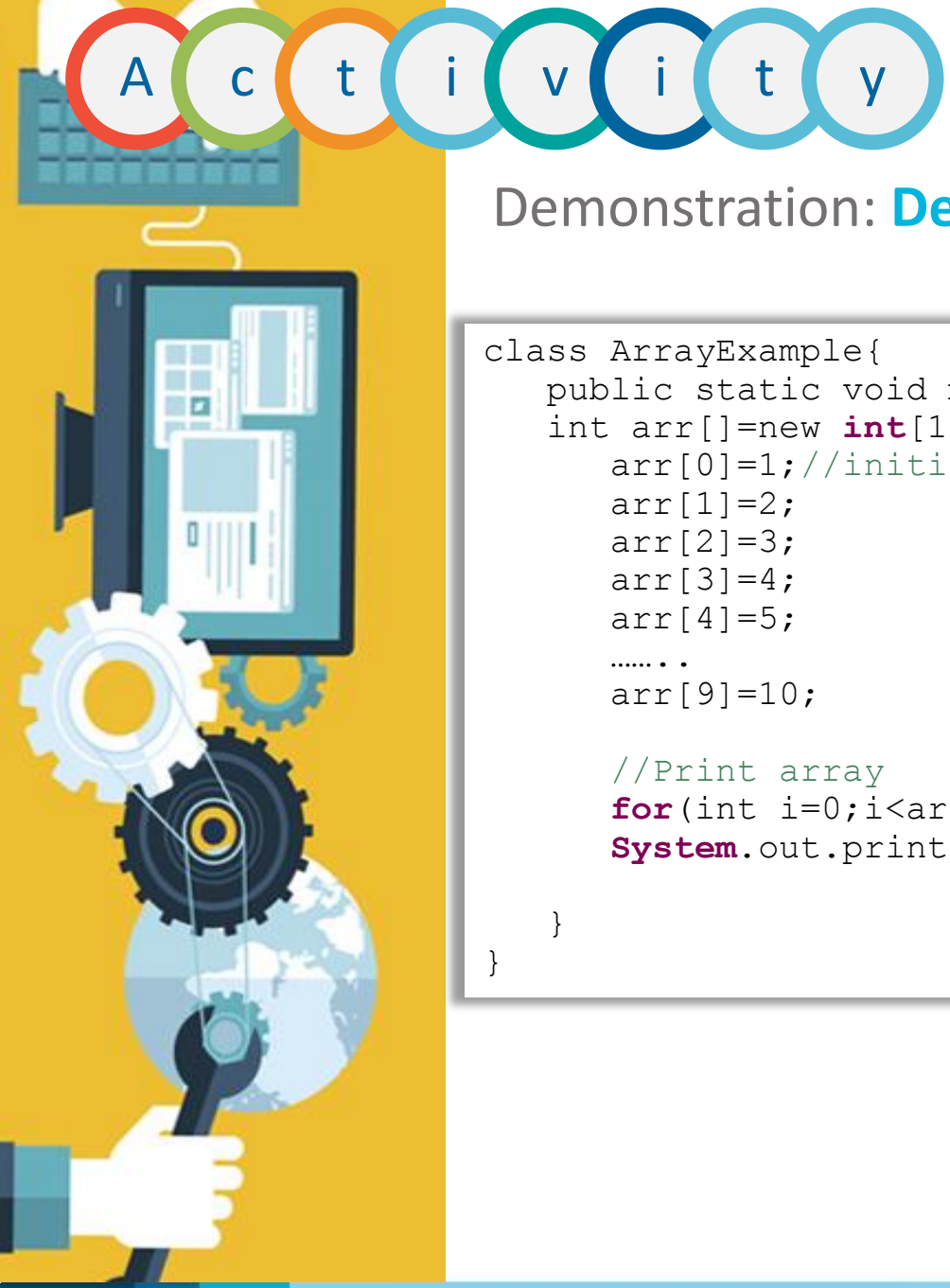
B

You need to store multiple values of the same type.

C

You want to avoid creating a new string each time the same value is used.

IBM



Brighter Blue

Demonstration: **Declaration of arrays**

```
class ArrayExample{
   public static void main(String args[]){
   int arr[]=new int[10];//declaration and instantiation
      arr[0]=1;//initialization
      arr[1]=2;
      arr[2]=3;
      arr[3]=4;
      arr[4]=5;
      ……..
      arr[9]=10;

      //Print array
      for(int i=0;i<arr.length;i++)//Length of array
      System.out.println(arr[i]);

   }
}
```

Creating arrays

Use the **new** keyword to create and initialize a primitive array.

```
public char[] createArray()
 {
                char[] s;
                s = new char[26];
                for ( int i=0; i<26; i++ )
                {
                s[i] = (char) ('A' + i);
                }
                return s;
 }
```

An example to create and initialize a **primitive (char) array**

Multi-dimensional arrays

A multi-dimensional array is an **array of arrays**.

The first call to **new** creates an object, an array that contains four elements.

```
int [] [] twoDim = new int [4] [];
twoDim[0] = new int [5];
twoDim[1] = new int[5];
```

Array bounds

The following code uses the **length** attribute to iterate on an array:

```
public void printElements(int[] list)
        {
                for (int i = 0; i < list.length; i++)   {
                  System.out.println(list[i]);
                }
            }
```

Array **resizing**

You **cannot resize** an array.

```
int[] myArray = new int[6];
        myArray = new int[10];
```

The same reference variable can be used to refer to an entirely new array

Copying arrays

- The Java programming language provides a special method in the System class, **arraycopy()** , to copy arrays.

- For example:

```
int myarray[] = {1,2,3,4,5,6}; // original array
int hold[] = {10,9,8,7,6,5,4,3,2,1}; // new larger array
System.arraycopy(myarray,0,hold,0,myarray.length);
// copy all of the myarray array to the hold array, starting with the 0th index
```

The contents of the array hold will be:

1,2,3,4,5,6,4,3,2,1

Spot Quiz

Brighter Blue

03

Which four options describe the correct default values for array elements of the types indicated?

1.int -> 0

2.String -> "null"

3.Dog -> null

4.char -> '\u0000'

5.float -> 0.0f

6.boolean -> true

A. 1, 2, 3, and 4

B. 1, 3, 4, and 5

C. 2, 4, 5, and 6

D. 3, 4, 5, and 6

Spot Quiz

Brighter Blue

04

Which will legally declare, construct and initialize an array?

A	int [] myList = {"1", "2", "3"};
B	int [] myList = (5, 8, 2);
C	int myList [] [] = {4,9,7,0};
D	int myList [] = {4, 3, 7};

Spot quiz

Brighter Blue

05

Which of the following are legal array declarations?

1. int [] myScores [];
2. char [] myChars;
3. int [6] myScores;
4. Dog myDogs [];
5. Dog myDogs [7];

A	1, 2, and 4
B	2, 4, and 5
C	2, 3, and 4
D	All of the above

Spot quiz

Brighter Blue

06

Which one of the following will declare an array and initialize it with five numbers?

A	Array a = new Array(5);
B	int [] a = {23,22,21,20,19};
C	int a [] = new int[5];
D	int [5] array;

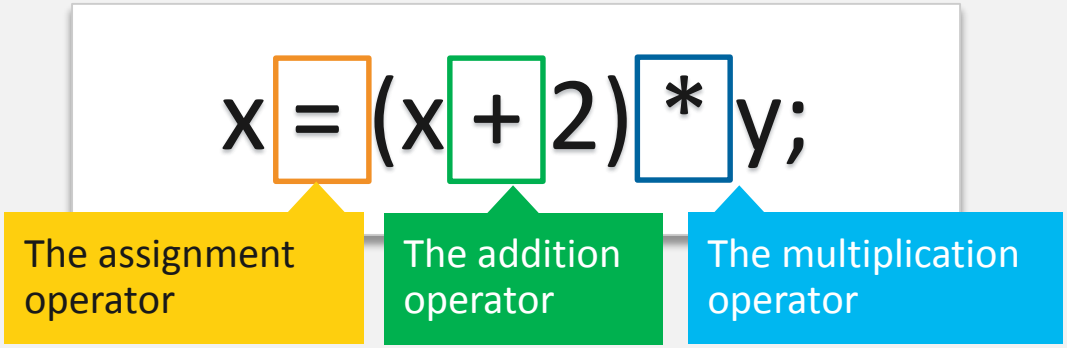
Brighter Blue

06 Overview of Java Operators

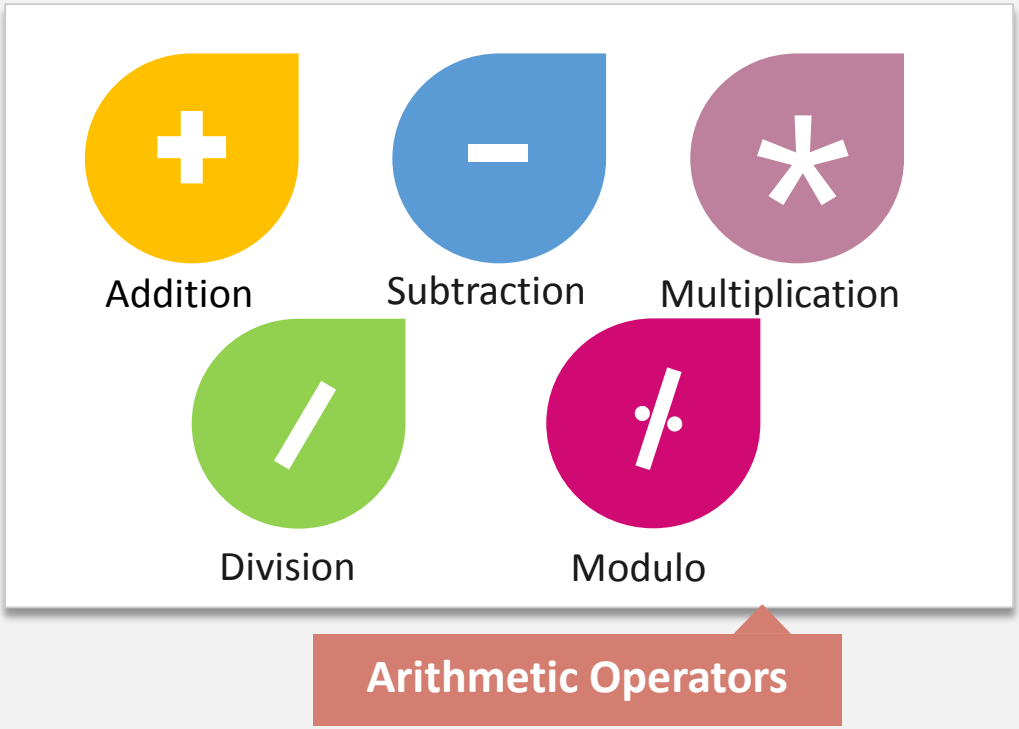
IBM

Components of an **expression**

The expression given below contains three operators:



What is an **arithmetic operator**?



Java supports arithmetic operators for integer and floating-point numbers.

numericOperand1 arithmeticOperator numericOperand2

Using arithmetic operators

- In this example, the code concatenates the strings **x** and **y** whose content is 2 and 3 respectively, so the output is a string, "**23**".

```
String x = "2";
String y = "3";
String z = x + y;
```

- In this example, only one of the operands is a string :

```
String x = "Result of multiplication operator :";
int result = 12;
String z = x + result;
```

The Result variable is converted to a string and the output is "**Result of multiplication operator: 12**".

Modulo operator

Here, the **left operand** is negative, so the result is also negative.

```
int firstInt = -5;
int secondInt = 2;
int myResult = firstInt % secondInt;
```

The value of myResult is -1

Using **arithmetic operators** with **primitive data types**

Brighter Blue

For example, consider the code that assigns the product of **a** and **b** to **a**.

```
short a;
short b;
a = b = 2;
a = a * b;
System.out.println (a);
```

This code will **not** compile because the result of the operation **a * b** is an **int**, not a short, so it cannot be assigned to 'a' automatically.

What is a **unary operator**?

```
numericOperand++

// is the equivalent of

numericOperand = numericOperand + 1;
```

The new value is **assigned back** to the operand, so the operand must be a variable.

Operator's **postfix version**

- Consider 6++
- This is not a valid operation because, it is equivalent to saying 6 = 6 + 1.
- In this example, the expression evaluates to the value of the operand before it is incremented. This is known as the **operator's postfix version**.
- The variable x is incremented to 7 after the expression evaluates.

```
int x = 6;
int y = x++;
```

In this code, the value assigned to **y** is **6**, not **7.**

Operator's **prefix version**

- You can also place an increment or decrement operator **before an operand**.
- The expression now evaluates to the operand's value after the increment or decrement operation has been performed. This is called the **operator's prefix version**.
- So, in the code, the value assigned to y is 7, not 6.

```
int x = 6;
int y = ++x;
```

In this code, the value assigned to **y** is **6**, not **7.**

Activity

Brighter Blue

Demonstration: **Prefix – Postfix**

```
class TestPrePost {
   public static void main(String[] args){
      int i = 6;
      i++;
      System.out.println(i);
      ++i;
      System.out.println(i);
      System.out.println(++i);
      System.out.println(i++);
      System.out.println(i);
   }
}
```

Spot Quiz

Brighter Blue

01

Suppose you want to build a valid Java expression that will compute the remainder as the result. Given the preliminary code, which expression will compute the remainder correctly?

```
short g;
short h;
int i;
int j;
int k;
g = h = 3;
i = j = k = 1;
```

A g = g * h;

B k = i / j--;

C k = j % i--;

D k = j % --i;

IBM

Using **unary operators** to **reverse signs**

Brighter Blue

- The unary minus (-) operator can be used to arithmetically **reverse the sign of a number or expression** to the right of the operator.
- If a is negative, the - operator changes the sign to positive.
- Also, the unary plus (+) operator can be used to return the value of the operand after Java's arithmetic promotion, in which byte, short and char values are promoted to int values.
- The + operator has **no "reversing" effect** on the sign of a number or expression.

Spot Quiz

02

After the given code executes, what is the value of variable c?

```
int a = 5;
int b = 3;
int c = a++ + --b;
```

A 7

B 8

C 9

Actions of **relational operators**

Relational Operator	Description
>	The > operator tests if the left-hand value is greater than the right.
>=	The >= operator tests if the left-hand value is greater than or equal to the right.
<	The < operator tests if the left-hand value is less than the right.
<=	The <= operator tests if the left-hand value is less than or equal to the right.
==	The equal to (==) operator tests equality and evaluates to true when two values are equal.
!=	The not equal to (!=) operator tests inequality and evaluates to true if the two values are not equal.

Action of **conditional operator**

Conditional Operator	Description

?:	The ternary operator **takes three operands**. The first operand is **boolean**, and the other two can be of any type. If the boolean operand is true, the result of the expression is the second operand; if it is false, the result is the **third** operand.
	
	In the code example, c will be assigned a value of 1 because b has a value of true.
	
	`boolean b;`
	`int c;`
	`b = true;`
	`c = (b ? 1 : 2);`

Brighter Blue

Demonstration: **Conditional Operator**

```
public class TestCondition {

   public static void main(String args[]){
      int a , b;
      a = 1;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );

      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
}
```

Spot Quiz

Brighter Blue

03

Given the preliminary code, which expressions will evaluate to true?

```
int a = 10;
int b = 20;
int c = 30 - b;
int d = c - a;
```

A	a <= b;
B	b == (c + 10);
C	c >= b;
D	d != a;

Actions of **logical operators** (1 of 3)

Logical Operator	Description
&&	AND (**&**) operator, evaluates the first operand and, if that is false, it returns a value of **false without evaluating the second operand**. But if the first operand is true, the operator **evaluates the second operand to check if it is also true** before it returns a value of true. For example, if **b** was declared as a boolean type, the expression would return a value of **false to b**, because the first operand evaluates to false. The expression is: `b = ( 100 < ( 5 * 6 ) ) && ( 100 > ( 8 * 8 ) );`

Actions of **logical operators** (2 of 3)

Logical Operator	Description
||	The || operator is a **"short circuit" version of the OR (|) operator** that speeds up the evaluation of compound boolean expressions. It evaluates the first operand and, if that is true, the operation returns a true value without evaluating the second operand. If the first operand is false, the || operator evaluates the second operand to determine if that is true before it returns a value of true. For example, if b was declared as a boolean type, the expression would return a value of true to b, because the second operand evaluates to true. The expression is: `b = ( 100 < ( 5 * 6 ) ) || ( 100 > ( 8 * 8 ) );`

Actions of **logical operators** (3 of 3)

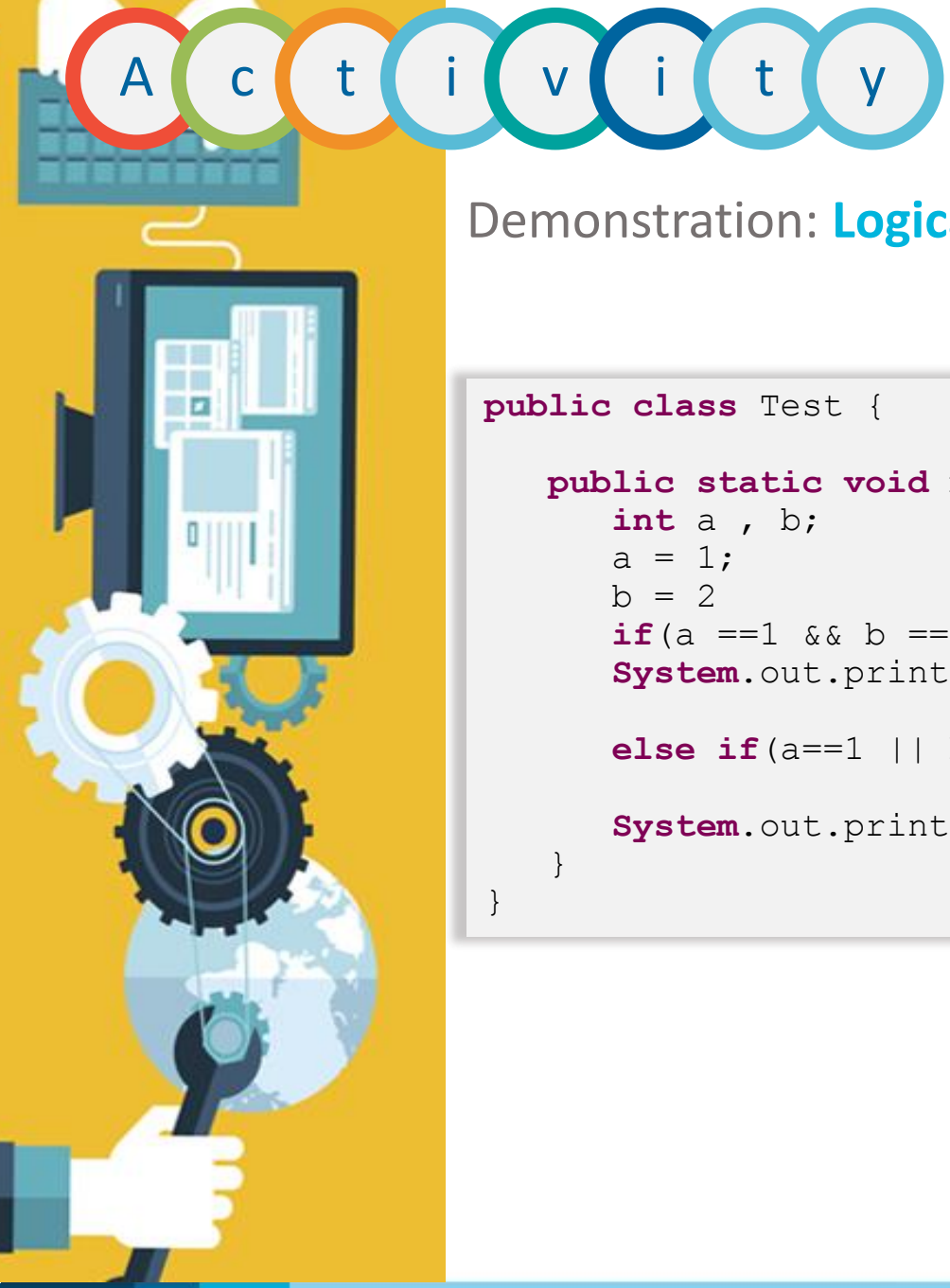
The boolean complement, or NOT, operator is the only unary boolean operator. It **returns a value of true if the operand is false**, and a **value of false if the operand is true**.

For example, the complement value of !false in the expression is true. The expression is:

```
(!false == true)
```

! You **should not use** short circuit operators in situations in which you require a side effect of the evaluation of the second operand to occur.
For example, in the code, the first operand is false, so the second operand is not evaluated and the increment does not take place, a still has a value of 4. If the program logic depended on this increment, you would need to use the AND (&) operator instead of the short circuit version.

```
int a;
int b;
a = b = 4;
if ((a !=b) && (a++ ==4)){
//do something
}
System.out.println ("The value of a is " + a);
```



Brighter Blue

Demonstration: **Logical Operators**

```
public class Test {

   public static void main(String args[]){
      int a , b;
      a = 1;
      b = 2
      if(a ==1 && b ==1)
      System.out.println("Value of a=1 and b= 1");

      else if(a==1 || b =1)

      System.out.println("Value of either of a and b is 1");
   }
}
```

Spot quiz

Brighter Blue

04

What will be the output of the program?

```
class PassA
    public static void main(String [] args) {
        PassA p = new PassA();
        p.start();
    }
    void start() {
        long [] a1 = {3,4,5};
        long [] a2 = fix(a1);
        System.out.print(a1[0] + a1[1] + a1[2] + " ");
        System.out.println(a2[0] + a2[1] + a2[2]);
    }
    long [] fix(long [] a3) {
        a3[1] = 7;
        return a3;   }
```

A	12 15
B	15 15
C	3 4 5 3 7 5
D	3 7 5 3 7 5

IBM

Spot quiz

05

What will be the output of the program?

```
class Test {
    public static void main(String [] args) {
        int x= 0;
        int y= 0;
        for (int z = 0; z < 5; z++) {
            if (( ++x > 2 ) && (++y > 2)) {

                x++;
            }
        }
        System.out.println(x + " " + y);
    }
}
```

A 5 2

B 5 3

C 6 3

D 6 4

Spot quiz

Brighter Blue

06

What will be the output of the program?

```
class Test {
    public static void main(String [] args) {
        int x= 0;
        int y= 0;
        for (int z = 0; z < 5; z++) {
            if (( ++x > 2 ) || (++y > 2)) {

                x++;
            }
        }
    System.out.println(x + " " + y);
    }
}
```

A	5 3
B	8 2
C	8 3
D	8 5

IBM

Spot quiz

Brighter Blue

07

What will be the output of the program? Which two are acceptable types for x?

1.byte
2.long
3.char

4.float
5.short
6.Long

```
switch(x)
{
    default:
        System.out.println("Hello");
}
```

A 1 and 3

B 2 and 4

C 3 and 5

D 4 and 6

Spot quiz

Brighter Blue

08

Which statement is true?

```
public void test(int x)
{
    int odd = 1;
    if(odd) /* Line 4 */
    {
        System.out.println("odd");
    }
    else
    {
        System.out.println("even");
    }
}
```

A Compilation fails.

B "odd" will always be output.

C "even" will always be output.

D "odd" will be output for odd values of x, and "even" for even values.

Brighter Blue

07 Expressions and Flow Control in Java

IBM

What is a **Java expression**?

A Java *expression* is a construct made up of variables, operators, and method invocations.

```
int result = 1 + 2; // result is now 3

if (value1 == value2)

    System.out.println("value1 == value2");
```

You can construct compound expressions from various smaller expressions.

- **Example 1:** **1 * 2 * 3** — Outcome always remains the same
- **Example 2**: **1 + 2 / 100** — Ambiguous as to which operation to be performed first
- **Example 3**: **(1 + 2) / 100** — Unambiguous and recommended

Simple **if** statements

The **if statement** syntax is given here:

```
if ( <boolean_expression> )
<statement_or_block>
```

```
if ( x < 10 )

      System.out.println("Are you finished yet?");

or (recommended):

if ( x < 10 ) {

System.out.println("Are you finished yet?");

}
```

Example: if statement

Complex **if-else** statements

The **if-else statement** syntax is given here:

```
if ( <boolean_expression> )

<statement_or_block>

else

<statement_or_block>
```

```
if ( x < 10 ) {

        System.out.println("Are you finished yet?");

} else {

        System.out.println("Keep working...");

}
```

Example: if-else statement

If-else-if statements

The if-else-if statement syntax is given here:

```
if ( <boolean_expression> )

<statement_or_block>

else if ( <boolean_expression> )

<statement_or_block>
```

```
int count = getCount();// a method defined in the class
if count (count<0){
   System.out.println("Error: count value is negative.");
   } else if (count>getMaxCount()){
   System.out.println("Error: count value is too big.");
   } else {
   System.out.println("There will be + count + people for lunch today.");
}
```

Example: if-else-if statement

Nested if…else statement

One if or else if statement can be used inside another if or else if statement.

```
if (condition 1) {      //begin loop
1
if (condition 2) {      //begin loop
2
if (condition 3) {      //begin loop
3
...
}
else {
...
}//end loop 3
}
else {
...
}//end loop 2
}
else {
...
}//end loop 1
```

Syntax for nested if…else statement

Switch statement

A switch statement allows a variable to be tested for equality against a list of values.

```
switch ( <expression> ) {
   case <constant1>:
    <statement_or_block>*
    [break;]
   case <constant2>:
    <statement_or_block>*
    [break;]
  default:
   <statement_or_block>*
   [break;]
}
```

Syntax for switch statement

Spot quiz

Brighter Blue

01

You are applying a switch statement in your code. What should be the value for the case?

A. The data type of the case should be constant.

B. The data type of the case should be same as in the switch.

C. The data type of the case should be integral.

IBM

Types of **Java loops**

Java has three **looping mechanisms** that are flexible.

A **while** loop

A **while** loop is a control structure that allows you to repeat a task a certain number of times.

```
int i = 0;
while ( i < 10 ) {
   System.out.println(i + " squared is " + (i*i));
   i++;
}
```

Example: While Loop

A **do...while** loop

A **do...while** loop is similar to a **while** loop, except that a do...while loop is guaranteed to execute at least one time.

```
int  i = 0;
do {
   System.out.println(i + " squared is " + (i*i));
   i++;
} while ( i < 10 );
```

Example: do…while loop

A **for** loop

A **for** loop is useful when you know how many times a task is to be repeated.

```
for ( int i = 0; i < 10; i++ )
System.out.println(i + " squared is " + (i*i));
```

Example: for loop

Spot quiz

Brighter Blue

02

You need to repeat a task at least once. Which loop would you add in your program?

A	A **while** loop.
B	A **for** loop.
C	A **do…while** loop.

IBM

What is a **break statement**?

A break statement is used to prematurely exit from switch statements, loop statements, and labeled blocks.

```
do {
    statement;
    if ( condition ) {
     break;
    }
     statement;
} while ( test_expr );
```

Example: Break Statement

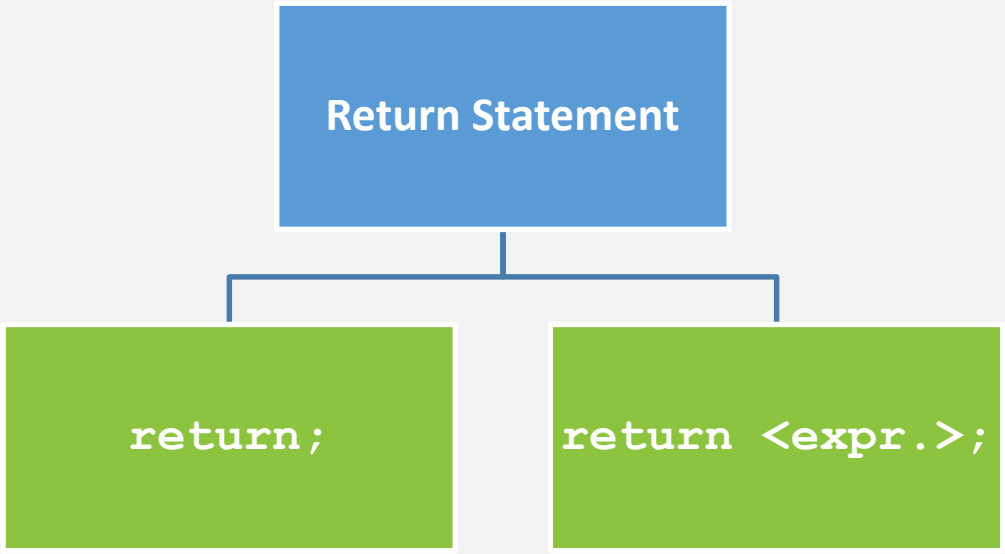
What is a **continue statement**?

A continue statement is used to skip over and jump to the end of the loop body.

```
do {
    statement;
    if ( condition ) {
     continue;
    }
     statement;
} while ( test_expr );
```

Example: Continue Statement

What is a **return statement**?



What is a **block statement**?

A block statement is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

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

Example: Java Block Statement

Spot quiz

Brighter Blue

03

You want to exit from switch and loop statements. What is the code you will use for this purpose?

A	Continue statement
B	Return statement
C	Break statement

IBM

Brighter Blue

08 Classes and Objects

IBM

OO SDLC phases

Identifying classes and their relationships

Designing the data members and methods of a class

Implementing the design using an OOP language

Java **class** and **modifiers**

Java Class

Classes are **templates** that are used to **create objects**, and to **define object data types** and **methods**.

Modifiers are **keywords** placed in a class, method, or variable declaration that changes how it operates.

The modifier precedes the rest of the statement.

Java Modifiers

Declaring **Java classes**

The basic syntax of a Java class is given below:

```
<modifier>* class <class_name>
{       <attribute_declaration>*
        <constructor_declaration>*
        <method_declaration>*
}
```

Example of a Java class:

```
public class MyFirstClass
{ private int age;
        public void setAge(int value)
        { age = value;
        }
}
```

Declaring **attributes**

The basic syntax of an attribute is given below:

```
<modifier>* <type> <name> [ = <initial_value>];
```

Example of an attribute:

```
public class MyFirstClass
{
   private int x;
   private float y = 10000.0F;
   private String name = "IBM";
}
```

Declaring **methods**

The basic syntax of a method is given below:

```
<modifier>* <return_type> <name> ( <argument>* ){    <statement>*    }
```

Example:

```
public class PolicyHolder
{
    private int policyNo;
    public int getPolicyNo()
    {    return policyNo;
    }
    public void setPolicyNo(int newPolicyNo)
    {    if (newPolicyNo > 0)
         {    policyNo = newPolicyNo;
    }   } }
```

Accessing **object members**

The dot notation is given below:

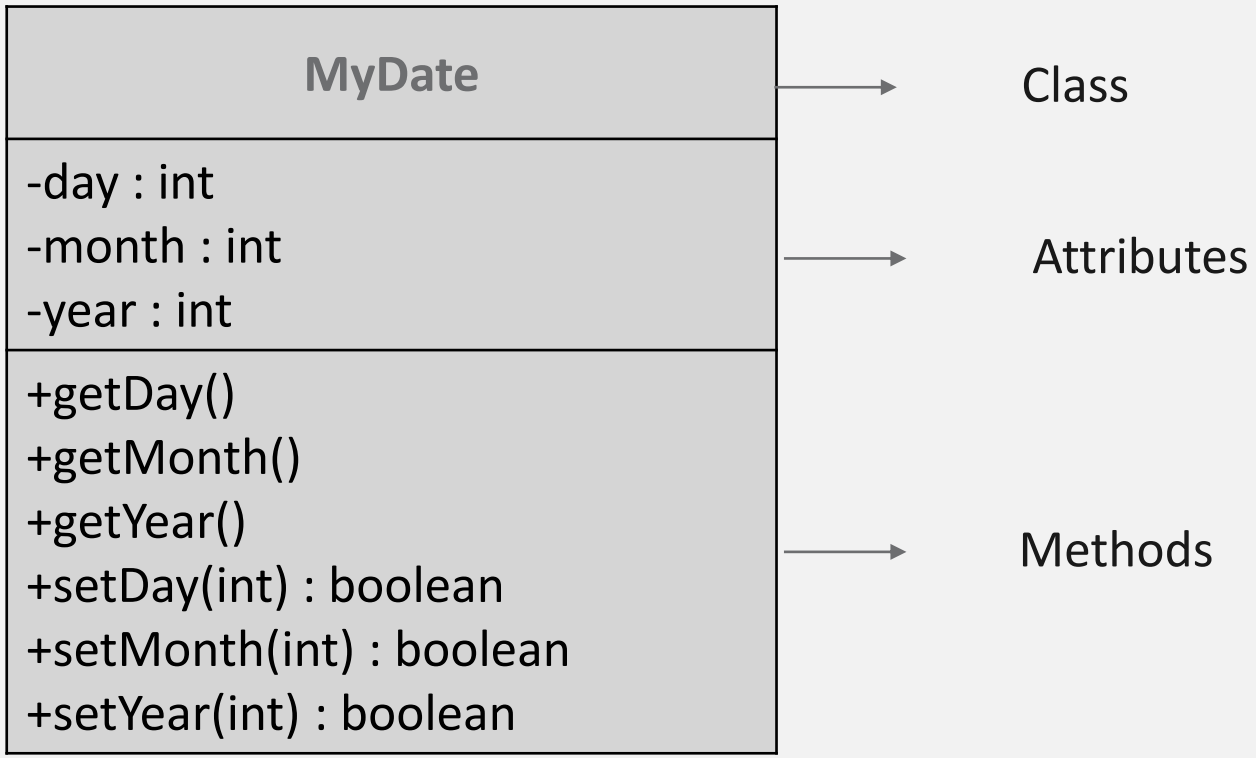
```
<object>.<member>
```

Example:

```
p.setPolicyNo(10000123)
p.policyNo=10000123

System.out.println("Print Policy No :"+
p.getPolicyNo())
will display Print Policy No : 10000123
```

Class representation **using UML** (Unified Modeling Language) Notation

Class representation of the MyDate class:



Reference variables in Java

A **reference** variable refers to a class name and helps to access a given object.

- Syntaxes to create a reference to an int array:

```
int x[];
int [] x;
```

- The reference x can be used for referring to any int array:

```
//Declare a reference to an int array
int [] x;
//Create a new int array and make x refer to it
x = new int[5];
```

- Example:

```
Policy p;
p = new Policy();
```

Classes in Java

Brighter Blue

Data members (state)

```
public class PolicyHolder{
        private int policyNo;
        private double bonus;
        //Other Data Members
        public void setPolicyNo(int no){policyNo = no;}
        public int getPolicyNo(){return policyNo;}
        public void setBonus(char amount){bonus = amount;}
        public double getBonus(){return bonus;}
        //Other Methods
}
```

1

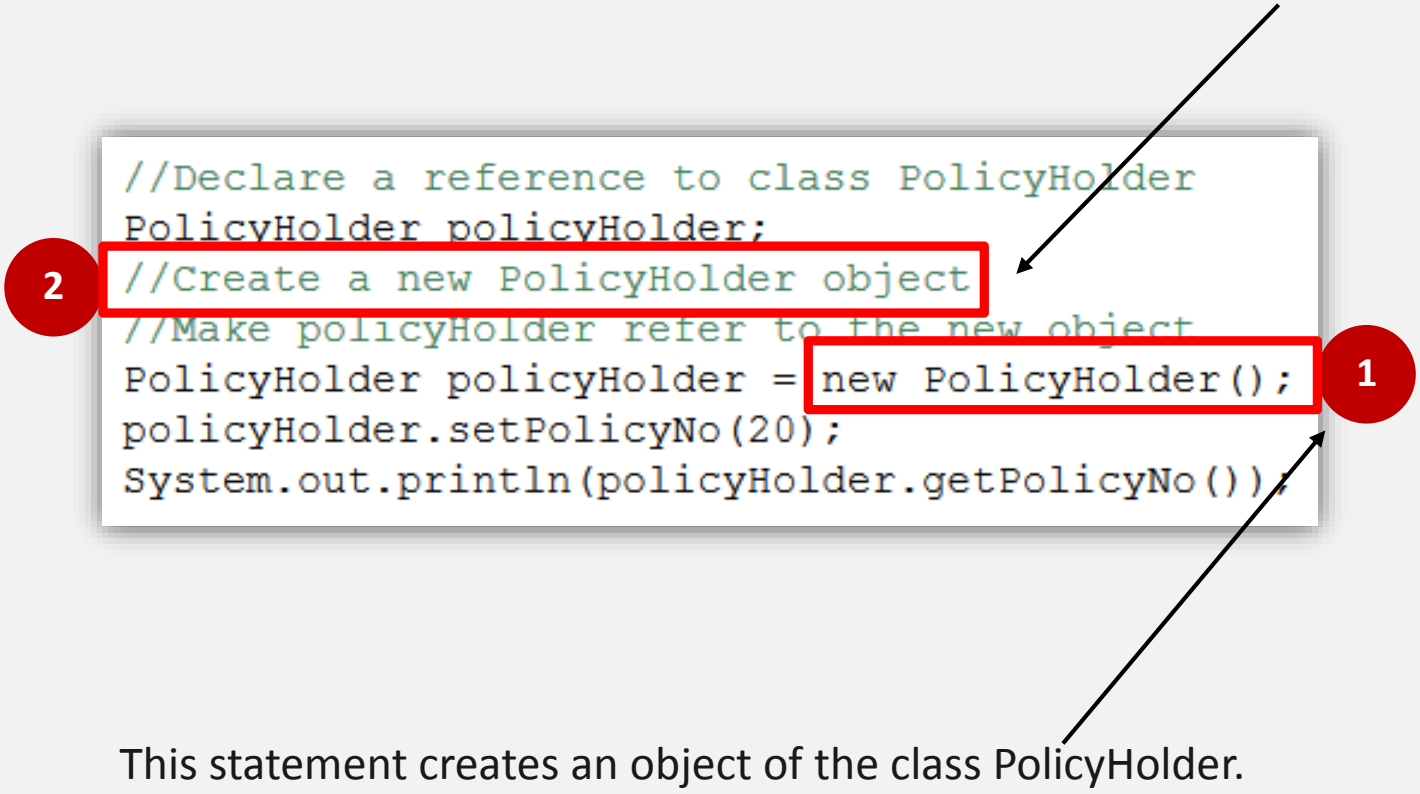
2

Methods (behavior)

The main method may or may not be present in a class depending on whether it is a starter class or not.

Creating objects in Java

This statement creates a reference to the class PolicyHolder.



This statement creates an object of the class PolicyHolder.

Invoking methods in a class

This statement creates a new PolicyHolder object and assigns its reference to policyHolder.

```
//Declare a reference to class PolicyHolder
PolicyHolder policyHolder;
//Create a new PolicyHolder object
//Make policyHolder refer to the new object
PolicyHolder policyHolder = new PolicyHolder();
policyHolder.setPolicyNo(20);
System.out.println(policyHolder.getPolicyNo());
```

1

2

All the public members of the object can be accessed with the help of the reference.

The **this** reference

```
public int getPolicyNo(){
        return policyNo;
        //return this.policyNo
        //these statements are similar
}
```

```
public class PolicyHolder{
        private int policyNo;
        private double bonus;
        //Other Data Members
        public void setPolicyNo(int policyNo){this.policyNo = policyNo;}
        public int getPolicyNo(){return policyNo;}
        public void setBonus(char bonus){this.bonus = bonus;}
        public double getBonus(){return bonus;}
        //Other Methods
}
```

Spot quiz

Brighter Blue

01

What can be used only in conjunction with an object of a class?

A	A class method
B	A class variable
C	An instance method
D	An instance variable

IBM

Spot quiz

Brighter Blue

02

How do instance and class variables and methods function?

A	Class variables are shared among instances.
B	Instance variables contain specific values for each object of a class.
C	You can access class and instance variables in class methods.
D	You need an instance to use class variables and methods.

IBM

Spot quiz

Brighter Blue

03

Assume that a Customer object needs to place an order with an Order object. It needs to call the Order object's placeOrder method and pass it the number and type of items it wants to order.

What does the Customer object need to include in the message it sends to the Order object?

A

Pass Parameters representing the number and type of items being ordered to method placeOrder of Order object.

B

Pass Parameters representing the number and type of items being ordered to method placeOrder of Customer object.

C

Pass customer number to method placeOrder of Order object.

D

Pass customer number to method placeOrder of Customer object.

IBM

What is a **constructor**?

A constructor is a special method that is called to create a new object.

```
//Declare a reference to class PolicyHolder
PolicyHolder policyHolder;
//Create a new PolicyHolder object
//Make policyHolder refer to the new object
PolicyHolder policyHolder = new PolicyHolder();
policyHolder.setPolicyNo(20);
System.out.println(policyHolder.getPolicyNo());
```

User-defined constructors

Observe the slide to learn how to declare the constructor and create the object using the constructor.

```
public class PolicyHolder{
        //Data Members
        public PolicyHolder(){
                bonus = 100;
        }
        //Other Methods
}



PolicyHolder policyHolder = new PolicyHolder();
//policyHolder.bonus is initialized to 100
```

Method overloading

Overloading is the ability to define **more than one method** with the **same name** in a class. The compiler is able to distinguish between the methods because of their method signatures.

Calls to overloaded methods will be resolved during compile time using static polymorphism.

```
void print(int i){
        System.out.println(i);
}
void print(double d){
        System.out.println(d);
}
void print(char c){
        System.out.println(c);
```

Overloading the constructors

Like other methods, constructors also can be overloaded.

```
public class PolicyHolder{
        //Data Members
        public PolicyHolder(){
                bonus = 100;
        }
        public PolicyHolder(int policyNo, double bonus){
                this.policyNo = policyNo;
                this.bonus = bonus;
        }
        //Other Methods
}

PolicyHolder policyHolder1 = new PolicyHolder();
//policyHolder1.policyNo is 0 and bonus is 100
PolicyHolder policyHolder = new PolicyHolder(1, 200);
//policyHolder1.policyNo is 1 and bonus is 200
```

Spot quiz

Brighter Blue

05

Which of the following two methods would overload correctly?

1

```
void add (int a, int b)
void add (int a, float b)
void add (float a, int b)
void add (int a, int b, float c)
```

2

```
void add (int a, float b)
int  add  (int a, float b)
```

A Code 1

B Code 2

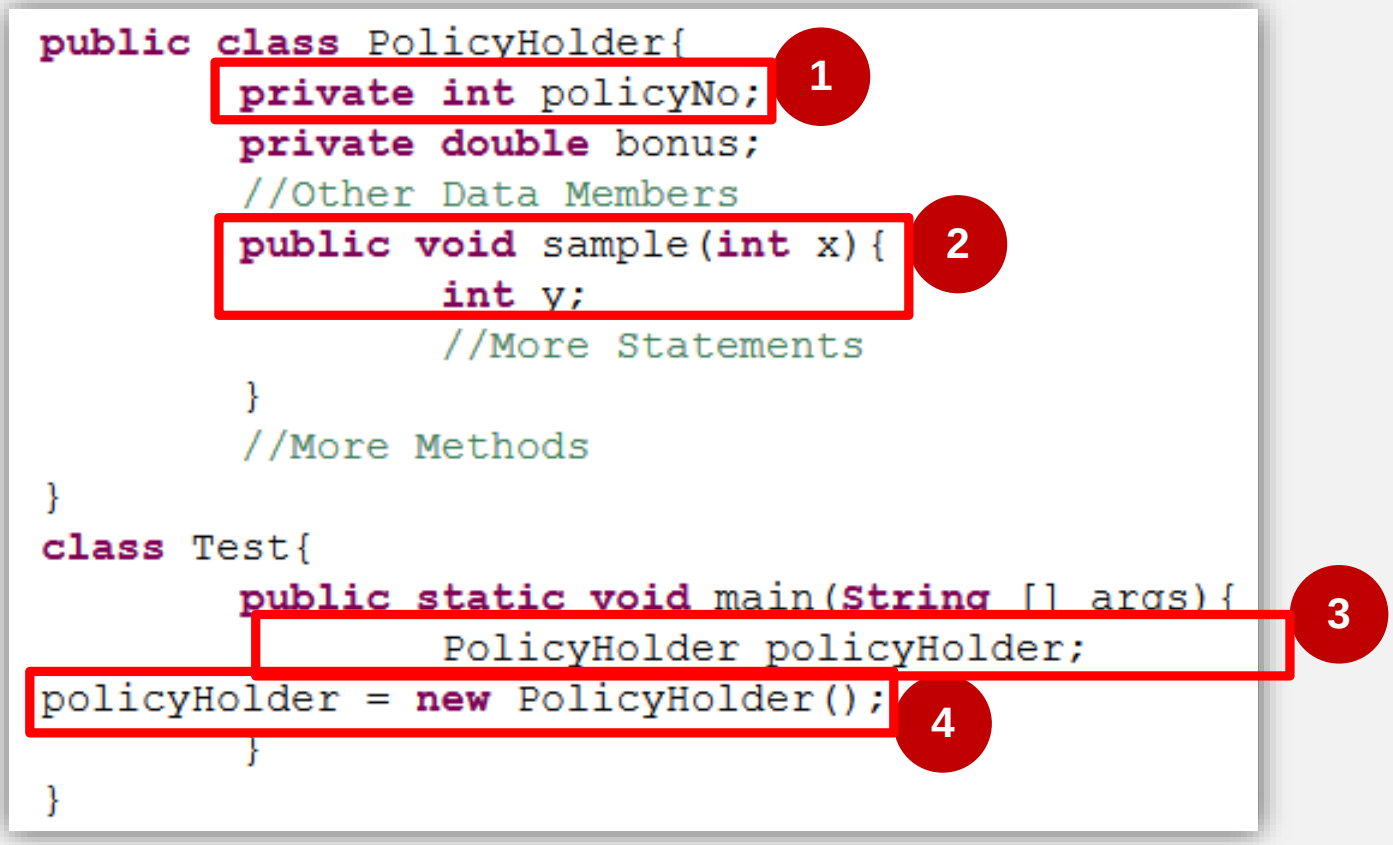
C Codes 1 and 2

D Neither of the codes

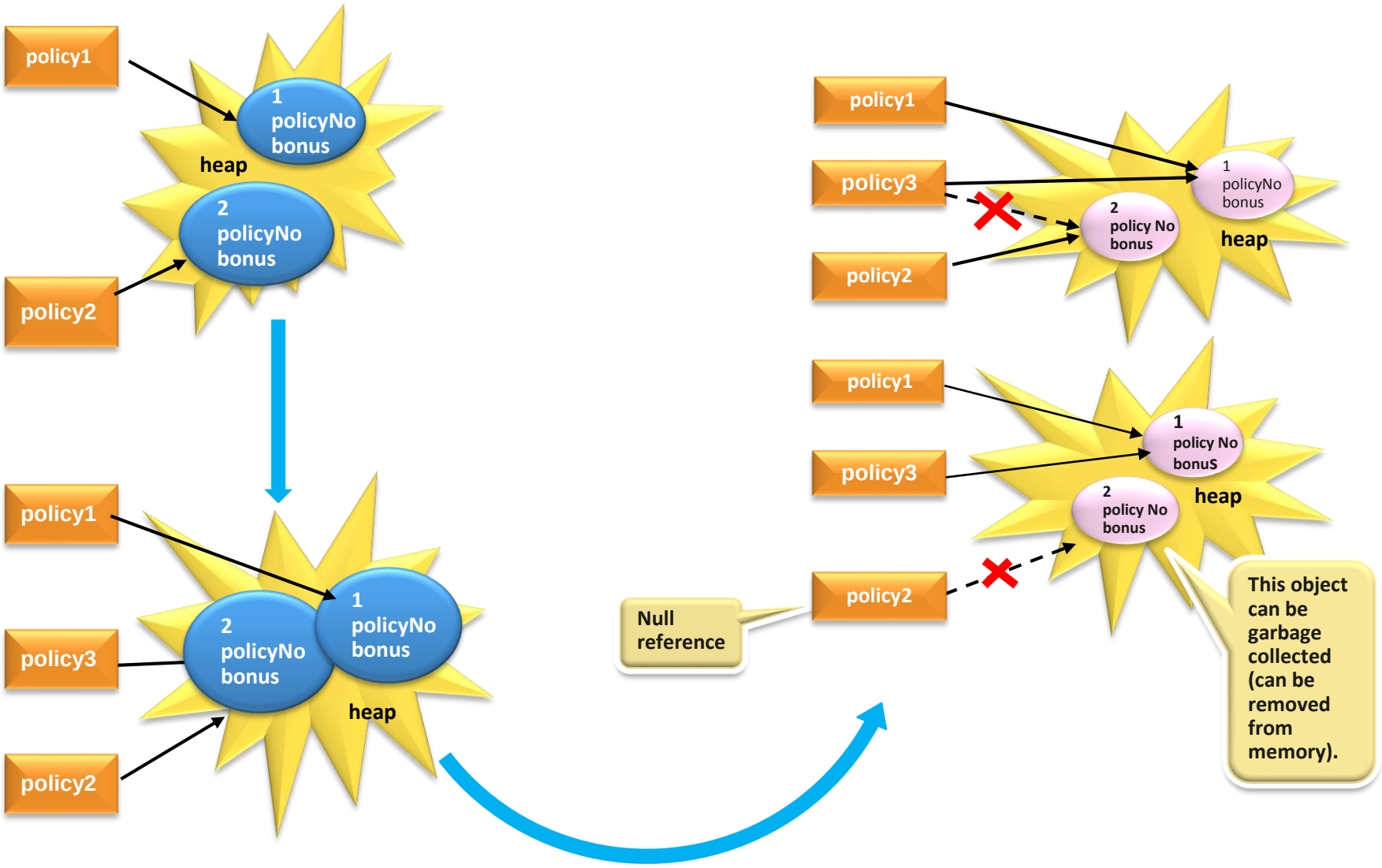
IBM

Memory allocation

Dynamically allocated arrays and objects are stored in a heap.



Lifetime of objects and garbage collection



Array of objects

```
PolicyHolder [] policyHolder = new PolicyHolder[3];
/*3 PolicyHolder references policyHolder[0], policyHolder[1] and policyHolder[2] are created and all 3 references are null*/
for(int i = 0; i < policyHolder.length; ++i){
        policyHolder[i] = new PolicyHolder();
        //Creating PolicyHolder object
}
for(int i = 0; i < policyHolder.length; ++i){
        System.out.println(policyHolder[i].getPolicyNo());
}
```

Object as **method arguments** and **return types**

```
public class PolicyHolder{
        //Data Members and Other Methods
        public boolean isSame(PolicyHolder policyHolder){
                return this.policyNo == policyHolder.policyNo;
        }
}

if (policyHolder1.isSame(policyHolder2)){
        //Statements
}
```

Static **data members**

```
public class PolicyHolder{
        private int policyNo;
        private double bonus;
        private static int total;
        //Other Data Members and Methods
}
```

The static variable is initialized to 0 only when the class is first loaded and not each time a new instance is made.

Static **methods**

```
public class PolicyHolder{
        private int policyNo;
        private double bonus;
        private static int total;
        //Other Data Members and Methods
        public PolicyHolder(){
                ++total;
                //Other Statements
        }
        public static int getTotal(){
                return total;
```

Each time the constructor is invoked and an object gets created, the static variable total will be incremented thus keeping a count of the total number of PolicyHolder objects created.

Static **methods** (continued)

Brighter Blue

Static methods are invoked using the syntax **<ClassName.MethodName>** :

```
System.out.println(PolicyHolder.getTotal());
//Prints 0
//A static method can be invoked without creating an object
PolicyHolder policyHolder1 = new PolicyHolder ();
System.out.println(PolicyHolder.getTotal());
//Prints 1
PolicyHolder policyHolder2 = new PolicyHolder ();
System.out.println(PolicyHolder.getTotal());
//Prints 2
```

Static block

The static block is a block of statement inside a Java class that will be executed when a class is first loaded.

Example:

```
class Test{
        static{
                //Code goes here
        }
}
```

Command line arguments

```
>java Sample Hello Welcome Ok
Hello, Welcome and Ok will be passed as an array of 3 elements to the main method of the class Sample
```

```
class Sample{
        public static void main(String [] args){
                for(int i = 0; i < args.length; ++i)
                        System.out.println(args[i]);
        }
}
```

Brighter Blue

09 Abstract Classes and Interfaces

IBM

What is an **abstract class**?

Abstract Class

A class **declared** with an **abstract** keyword is called an Abstract Class.
It can be **extended to implement** its methods.

An abstract class **can't be instantiated**.

- It facilitates reusability like any other base class: The data members and concrete methods of the abstract class can be reused.
- It defines a standard interface for a family of classes: The concrete sub classes have the abstract methods implemented.

What are the uses of an Abstract Class?

Abstract class : examples

```
public abstract class Policy {
    //Other Data and Methods
    public abstract double getBenefit();
}
```

The keyword **abstract**

```
public class TermInsurancePolicy extends Policy
{
    //Other Data and Methods
    public double getBenefit(){
        //Code goes here
    }
}
```

```
public class EndowmentPolicy extends Policy {
    //Other Data and Methods
    public double getBenefit(){
        //Code goes here
    }
}
```

The **getBenefit()** will not have any body

The **final** keyword

- Use all uppercase letters by convention.

```
final int NORTH = 1;
```

- The final methods cannot be overridden by the sub classes.

```
public final void sample (){
      //Method Definition
}
```

- A final class cannot be extended.

```
final class Test {
     //Class Definition
}
```

Spot quiz

Brighter Blue

01

The **final** keyword is used to restrict _____.

A Value change

B Method overriding

C Inheritance

D All of the above

What is an **interface**?

Brighter Blue

Interface

An interface is a **collection of abstract methods**. It looks like a class but is not a class.
An interface may have **methods** and **variables** that are just like the class, but they are **abstract** by default. The **variables** declared in an interface are **public, static**, and **final,** by default.

An interface body may contain:

- Abstract Methods: An abstract method within an interface is followed by a **semicolon (;),** but no braces.
- Default Methods: They are defined with the `default` modifier.
- Static Methods: They are defined with the `static` keyword.
- Constant Declarations: All the constant values defined are implicitly `public, static,` and `final.`

Spot quiz

Brighter Blue

02

Which of these is the valid declaration within an interface definition?

A

```
public double method();
```

B

```
public final double method();
```

C

```
static void method(double d1);
```

D

```
protected void methoda (double d1);
```


IBM

Abstract class versus Interface

Abstract Class	Interface
Can have abstract and non-abstract methods	Can have only abstract methods
Doesn't support multiple inheritance	Supports multiple inheritance
Can have final, non-final, static, and non-static variables	Has static and final variables only
Can have static methods, main method, and constructor	Can't have static methods, main method, and constructor
Can provide the implementation of interface	Can't provide implementation of abstract class
Can NOT be instantiated but they have subclasses	Can NOT be instantiated but you can have an instance of a class that implements the interface.

The abstract keyword is used to declare abstract class. Example:
`public abstract class Shape{`
`public abstract void draw();`
`}`
	The interface keyword is used to declare interface. Example:
	`public interface Drawable{`
	`void draw();`
	`}`

Abstract classes and interfaces

- Consider a case where the class ArrayList is extended from another class called Array.
- The class ArrayList cannot be extended from both Array and List at the same time. Java does not support multiple inheritance of classes.
- If the List is an interface, the problem can be solved as displayed:

```
class Arraylist extends Array, List {
        //will not compile
}
```

```
class ArrayList extends Array
implements List{
      //Members
}
```

Interfaces and inheritance

Just like how a class can be inherited from another class, an interface can be extended from another interface.

```
interface Editor {
      public void edit();
}
//ProgramEditor is an Editor
interface ProgramEditor extends Editor{
      public void highlightKeywords();
}
```

Implementing all interfaces

The class implementing EditorWithPainter should implement all the methods of Editor, Painter, and EditorWithPainter.

```
/*
The class implementing EditorWithPainter should
implement
all the methods of Editor, Painter, and
EditorWithPainter
*/
class DocumentCreator implements EditorWithPainter {
     public void edit(){
          //Code goes here
     }
     public void draw(){
          //Code goes here
     }
     public void wrapTextAroundPicture(){
          //Code goes here
}
```

Extending an interface

An interface can be extended from more than one interface.

```
interface Editor{
      public void edit();
}
interface Painter{
      public void draw();
}
interface EditorWithPainter extends Editor, Painter{
      public void wrapTextAroundPicture();
}
```

Implementing multiple interfaces

A class can extend only one class but implement any number of interfaces.

```
JavaIDE extends JavaCompiler implements Editor,
Helper{
      //Should implement all the methods of Editor
      //and Helper
      public void edit(){
            //Code goes here
      }
      public void displayHelp(){
            //Code goes here
      }
}
```

Typecasting of reference types

Assume that an EndowmentPolicy object is passed to the following method:

```
public void aMethod(Policy policy){
     policy.methodDecalreInPolicyClass();
     //policy.newMethodinEndowmentPolicyClass(); Error
     //EndowmentPolicy endowmentPolicy = policy; Error
     EndowmentPolicy endowmentPolicy =
(EndowmentPolicy)policy;
     endowmentPolicy.newMethodinEndowmentPolicyClass();
}
```

Passing an object of TermInsurancePolicy to the method will result in an error.

```
EndowmentPolicy endowmentPolicy = (EndowmentPolicy)policy;
//Error
//TermInsurancePolicy cannot be typecast to EndowmentPolicy
```

Spot quiz

Brighter Blue

03

Which of the following statements is / are **true**?

A	Abstract classes can have abstract and non-abstract methods.
B	Abstract classes do not support multiple inheritance.
C	Abstract classes can have static methods, main method, and constructor.
D	All of the above.

Spot quiz

Brighter Blue

04

Which one of the following statements is **true**?

A	Interfaces do not support multiple inheritance.
B	Interfaces can't have static methods, main method, or constructor.
C	Interfaces can't have abstract methods.
D	Interfaces can have any variables.

Brighter Blue

10 Packages in Java

IBM

What is a **package** in Java?

A package is a **collection** of related **classes** and **interfaces** in Java.

A group of related classes are bundled inside a package.

Examples of existing packages in Java:

- java.lang: bundles the fundamental classes
- java.io: bundles the classes for input-output functions

Why use packages in Java?

- To **avoid** naming conflicts
- To **control** access
- To **locate** classes, interfaces, enumerations, and annotations easily
- To **use** classes, interfaces, enumerations, annotations easily

Creating a package

- Create a folder called insurance.
- Place the program in this folder.
- Compile the program

```
package insurance;
public class Policy{
        //Code goes here
}
```

- The fully qualified name of a class that is stored in a package is given here:

```
<packagename>.<ClassName>
```

- The `<package name> .<Class name>` is given here :

```
insurance.Policy policy = new insurance.Policy();
```

Importing a class

The import statement can be used to import a class into a program so that the class name need not be fully qualified.

```
import insurance.Policy;
class Test{
        public static void main(String [] args){
                Policy policy = new Policy();
                //Policy means insurance.Policy
        }
}
```

Best Practice

Import only those classes that are required in your code; do not import the entire package.

Packages and classpath

- An environment variable classpath needs to be set.

```
>set classpath = %classpath%;C:\work\java
```

- The classpath points to a folder one level above the package folder.

```
package insurance.policy;
public class Policy{
        //Code goes here
}
```

Standard **Java packages**

Package	Description
`java.lang`	• Contains classes that form the basis of the design of the Java programming language • No need to explicitly import this package • Examples: String class, System class, and so on
`java.io`	• Contains classes and interfaces to perform I/O operations
`java.util`	• Contains utility classes and interfaces like list, calendar, and so on
`java.awt`	• Contains classes and interfaces to create GUI

Spot quiz

Brighter Blue

01

Which of these access specifiers can be used for a class, so that its members can be accessed by a different class in the same package?

A	Public	B	Protected
C	No Modifier	D	All of the mentioned

IBM

Spot quiz

Brighter Blue

02

Which of these access specifiers can be used for a class so that it's members can be accessed by a different class in the different package?

A Public

B Protected

C Private

D No Modifier

Spot quiz

Brighter Blue

03

Package declarations _____ import statements.

A	precede
B	succeed
C	precede or succeed
D	none

IBM

Java naming conventions

- Java naming convention is a set of commonly used rules to follow as you decide what to name your identifiers such as class, package, variable, constant, method etc.

Name	Convention
class name	Class names should be nouns in UpperCamelCase, with the first letter of every word capitalised. Use whole words — avoid acronyms and abbreviations e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	Methods should be verbs in lowerCamelCase or a multi-word name that begins with a verb in lowercase; that is, with the first letter lowercase and the first letters of subsequent words in uppercase. E.g actionPerformed(), println() etc.
variable name	should be in lowerCamelCase letter e.g. firstName, orderNumber etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc

Brighter Blue

Day 01 Practice Exercises:

1. Create a class Student with first name, last name, age, school, hobbies. Use private instance attributes. Public getters and setters. Create a test class to display the Student details.

2. Write a program that accepts a GBP value and converts it to USD. Assume currency exchange rate is 1USD = 0.66 GBP.

3. Write a program that counts the number of upper case letters in a String and outputs them. Use the Character API

4. Write a program that determines if a number is Odd or Even. Output to the console if even or odd.

5. Write a program that takes three command-line arguments: two integers followed by an arithmetic operator (+, -, * or /). The program shall perform the corresponding operation on the two integers and print the result. Sample output:

   java <Class_Name> 3 2 +
   3+2=5

IBM