The Java Series

Java Essentials I
What is Java?
Basic Language Constructs

What is Java?

A **general purpose** Object Oriented programming language.

Created by Sun Microsystems.

It s a general purpose language in exactly the same way as C, C++, FORTRAN, Pascal, are general purpose programming language.

As any general purpose language Java programs may be small applications, complex systems, etc

Why Java?

Cross-platform development is typically hard, implies: recompilations, adaptations, etc.

This is very inconvenient in:

- Heterogeneous networked environments.
- Environments with a large variety of machines.

Also, spread programming languages had things to improve:

- FORTRAN is not Object Oriented.
- C++ can become cumbersome to learn and debug.

Developed by Sun initially as a language to be commonly used in machines, pcs, tvs, vcrs, etc..

The goals

It had to be portable: Java programs should be able to run anywhere with no modifications.

It had to be Object Oriented, as OO is the accepted paradigm for sustainable software development

It had to learn from problems in other languages

It had to help programmers in their tasks.

It had to be network aware.

The Task

Sun designed the language from scratch borrowing much of C++ syntax.

It s purely Object Oriented. Everything in Java must be defined in some class.

Common functionality should be self-contained:

- General Purpose libraries (data manipulations, graphics, network, UI, DB, etc.) are part of the language distribution.

Programmers don t have to worry about internals of memory management.

Portability

Means that

- the same code should run anywhere, without the need to recompile, adapt, etc.
- when you develop you don t think about the platform your users will have.
- as a user, you don t need to worry about the platform used by developers.

Typically when you make a program you either:

- Compile the source code to obtain a machine specific binary which can be executed right away.
- Distribute the source and an interpreter in the user machine executes it.

Compilation vs Interpretation

- When compiling developers distribute binaries:
  - A binary is machine specific.
  - A binary is usually quite efficient because it s machine specific.
  - The compilation process checks for syntactic errors, optimizes code, etc.
- When interpreting, developers distribute source code:
  - Source code can be made machine independent.
  - Source code is not as efficient as binaries since there is an interpreter which: parses the source code and then executes it.

Portability

Always means a compromise between compilation and interpretation.

Java is neither compiled but interpreted but both

A Java program is a set of .java source files.

Java source files are compiled producing **Java Binaries** , called Byte Codes.

Byte Codes are machine independent. Programmers distribute compiled byte codes to their users.

To run applications, users MUST have a java byte-code interpreter for their machine. Byte-codes are therefore interpreted.

Java Portability

Portability is achieved at the byte codes.

There is a compilation process at the developers and an interpretation process at the user.

This way:

- Developers benefit from compilation (error checking, etc)
- Byte codes are more compact and optimized than source code.
- Portability is not lost which performance is far better than with typically interpreted languages.
- However, performance is (still) far from pure compiled languages.

JDK and JVM

A **Java Virtual Machine** understands and executes byte-codes.

In order to actually have portability there has to be a JVM for each different platform.

Any piece of software containing a JVM is able to execute Java applications.

Sun deploys Java by offering the **Java Development Kit** (jdk)

The JDK contains:

- A Java compiler to produce byte-codes from java source files.
- A Java interpreter to execute byte-codes.
- Libraries providing common functionality (DB, network, UI,..)
- Debugging utilities.

For instance, the programs: **`javac`** (the compiler) and **`java`** (the interpreter) contain a JVM.

Java and the WWW

Usually a user uses **`java`** from Sun s jdk to execute applications.

But some WWW browsers have included a JVM in their application. So they can execute byte-codes.

If byte-codes are available through a URL, they may be pointed to from within HTML

This way a browser may:

- Download an HTML page.
- Obtain an URL to a Java byte-code from an <APPLET> HTML tag. URL-pointed byte-codes are called **APPLETS**.
- Download the byte-code.
- Lend a piece of window to the JVM.
- Tell their JVM to execute the downloaded byte-code.

So the browser renders the HTML page and at the same time we have a java applet running.

For this, portability is essential.

Evolution of Java

Java is maintained and developed by Sun.

New versions of Java (improvements in the language, libraries, etc.) are made public through releases of the JDK.

Sun usually releases the JDK on Solaris and Windows. Other platforms follow as vendors implement it.

During Java evolution, JVM versioning has been quite an issue.

The current release of Java is quite stable and can be thought of THE ONE.

Remember OO

Abstraction + Decomposition + Organization

Steps:

- Define your classes
  - Properties (variables) + behaviour (methods)
  - Inheritance, Polymorphism, Encapsulation
- Create your instances from the classes definition.

We are going to go through these steps.

- You ll see how to use Java to do this.

A few things about Java

There is a set of atomic Data Types

int, boolean, etc.

Apart from that everything is an Object

There are operators (=, +, >=, etc.)

Control flow statements (if, while, etc.)

And a way to invoke operations on an object:

```
int size = 10;
myBox = new Box(size);
myBox.open();
myBox.printStatus();
```

Sample Scenario 1

We are going to be using counters.

Each counter has its own internal value (properties)

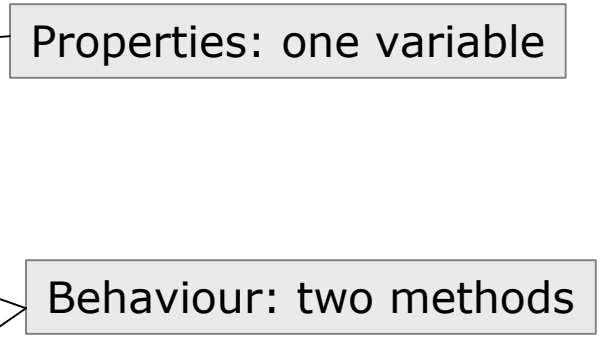
We can ask a counter to:

- Increment its value
- To print out its value

First we have to define a counter class

Scenario 1: Counter class

```
public class Counter {
	int value = 0;
	public int increment() {
		value ++;
		return value;
	}
	public int getValue() {
		return value;
	}
}
```



Scenario 1: MyApplication class

```
public class MyApplication {

    public static void main (String args[]) {

        Counter c1 = new Counter();
        Counter c2 = new Counter();

        System.out.println("Counter 1 has value "+c1.getValue());
        System.out.println("Counter 2 has value "+c2.getValue());

        c1.increment();
        c1.increment();
        c2.increment();

        System.out.println("Counter 1 now has value "+c1.getValue());
        System.out.println("Counter 2 now has value "+c2.getValue());

    }
}
```

Main method is STATIC
Specifies the program startup point

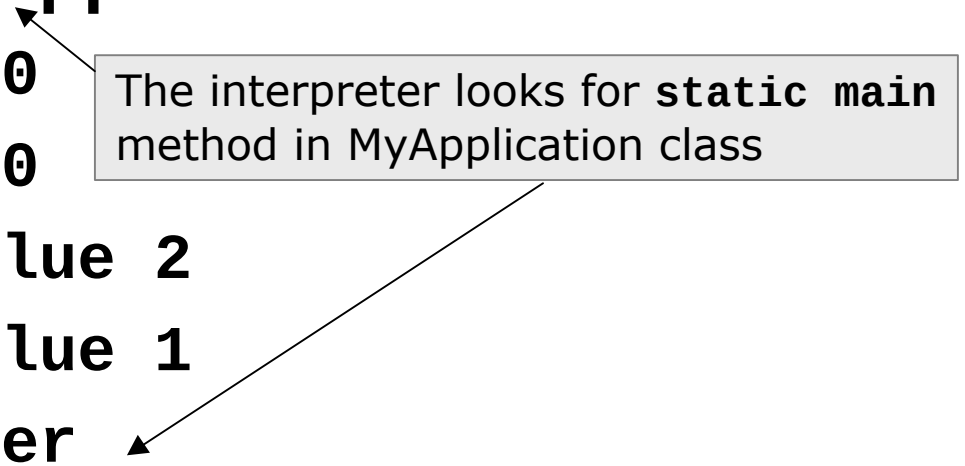
We create two objects

Invoke methods on objects

Invoke static class methods.
System is always available

Scenario1: Compiling and Executing

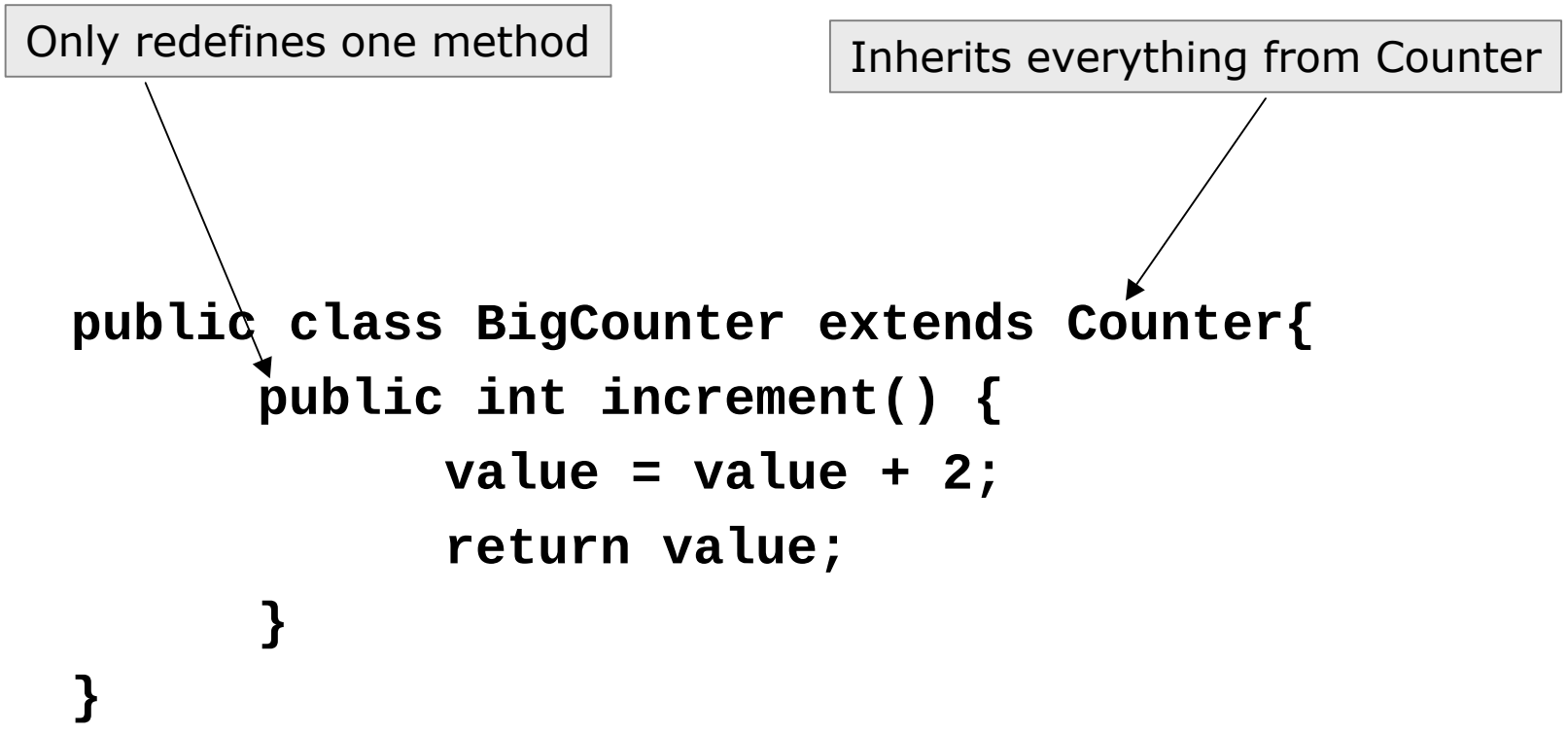
```
rsplus03> javac *.java
rsplus03> java MyApplication
Counter 1 has value 0
Counter 2 has value 0
Counter 1 now has value 2
Counter 2 now has value 1
rsplus03> java Counter
In class Counter: void main(String argv[]) is not defined
```



Sample Scenario 2

Now, we d also like to have a counter just as the one we have defined but each time the **`increment`** method is invoked we get the counter incremented by two.

Scenario 2: BigCounter class

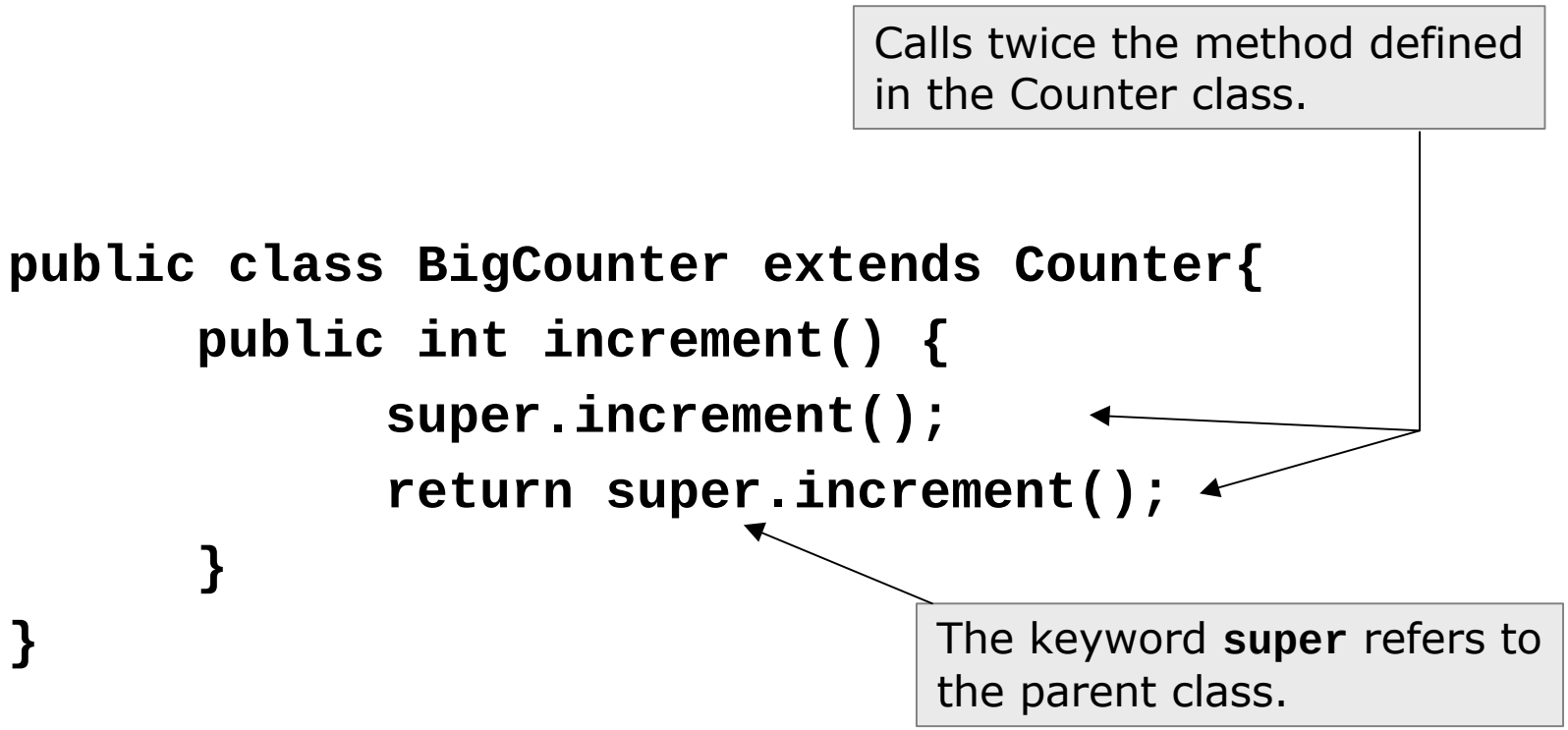


Sce 2:MyApplication class

```
public class MyApplication {

    public static void main (String args[]) {

        Counter c1 = new BigCounter();
        Counter c2 = new Counter();

        System.out.println("Counter 1 has value "+c1.getValue());
        System.out.println("Counter 2 has value "+c2.getValue());

        c1.increment();
        c1.increment();
        c2.increment();

        System.out.println("Counter 1 now has value "+c1.getValue());
        System.out.println("Counter 2 now has value "+c2.getValue());

    }
}
```

This is the ONLY CHANGE in the code

Sce 2: BigCounter class (2)



Sample Scenario 3

Now we want to optionally specify an starting value for counters when we create them.

Sce 3: Counter Class

We create a **constructor**

```
public class Counter {
        … …
        public Counter (int startingValue) {
                value = startingValue;
        }… …
}

public class BigCounter {
        public BigCounter (int value) {
                super(value);
        }
        … …
}
```

Constructors are NOT inherited. We must invoke the with **super()**

This constructor does whatever the parent s constructor does.

Sc 3: MyApplication Class

```
public class MyApplication {

    public static void main (String args[]) {

        Counter c1 = new Counter(3);
        Counter c2 = new BigCounter(4);

        System.out.println("Counter 1 has value "+c1.getValue());
        System.out.println("Counter 2 has value "+c2.getValue());

        c1.increment();
        c1.increment();
        c2.increment();

        System.out.println("Counter 1 now has value "+c1.getValue());
        System.out.println("Counter 2 now has value "+c2.getValue());

    }
}
```

Here we use the constructor

This is OO, we use indirectly the parent s constructor

Static Methods and Variables

A static method or variable is only attached to the class definition and to no object instantiated from that class.

Thus:

- They are accessed through the class name.
- All objects instantiated from that class see the same static variables.
- Static methods can be invoked directly from the class name.

Sce 4: Sample class

```
public class Sample {
    static int classValue = 0;
    int objectValue = 10;

    public static void printClassValue() {
        System.out.println(“The class value is “+classValue);
    }

    public static void setClassValue (int v) {
        classValue = v;
    }

    public void printObjectValue() {
        System.out.println(“This object’s value is “+objectValue);
    }
    public void setObjectValue(int v) {
        objectValue = v;
    }
};
```

Sce 4: MyApplication class

```
public class MyApplication {
    public static void main (String args[]) {
          Sample.printClassValue();     // prints 0;
          Sample s1 = new Sample();
          Sample s2 = new Sample();

          s1.setClassValue(20);
          s1.setObjectValue(25);
          s1.printClassValue();         // prints 20
          s2.printClassValue();         // prints also 20
          s1.printObjectValue();        // prints 25
          s2.printObjectValue();        // prints 10

          Sample.setClassValue(15);
          s2.printClassValue();         // prints 15
          s1.printClassValue();         // prints also 15
}
```

Scenario 5: Accumulator

```
public class Acumulator {
	int sum = 0;
	public void add (int quantity) {
		sum = sum + quantity;
	}
	public int getSum () {
		return sum;
	}
}
```

Scenario 5: MyApplication

```
public class MyApplication {

    public static void main (String args[]) {

        Accumulator a1 = new Accumulator();

        System.out.println(”Accumulator 1 has value "+a1.getSum());
        a1.add(5);
        System.out.println(”Accumulator 1 has value "+a1.getSum());
        a1.add(10);
        System.out.println(”Accumulator 1 has value "+a1.getSum());

    }
}
```

Scenario 5

But now we d like Counters and Accumulators to do something similar such as printing their value themselves.

We d like something like this for all objects:

```
Object.printStatus();
```

But preserving the structure of classes we have defined.

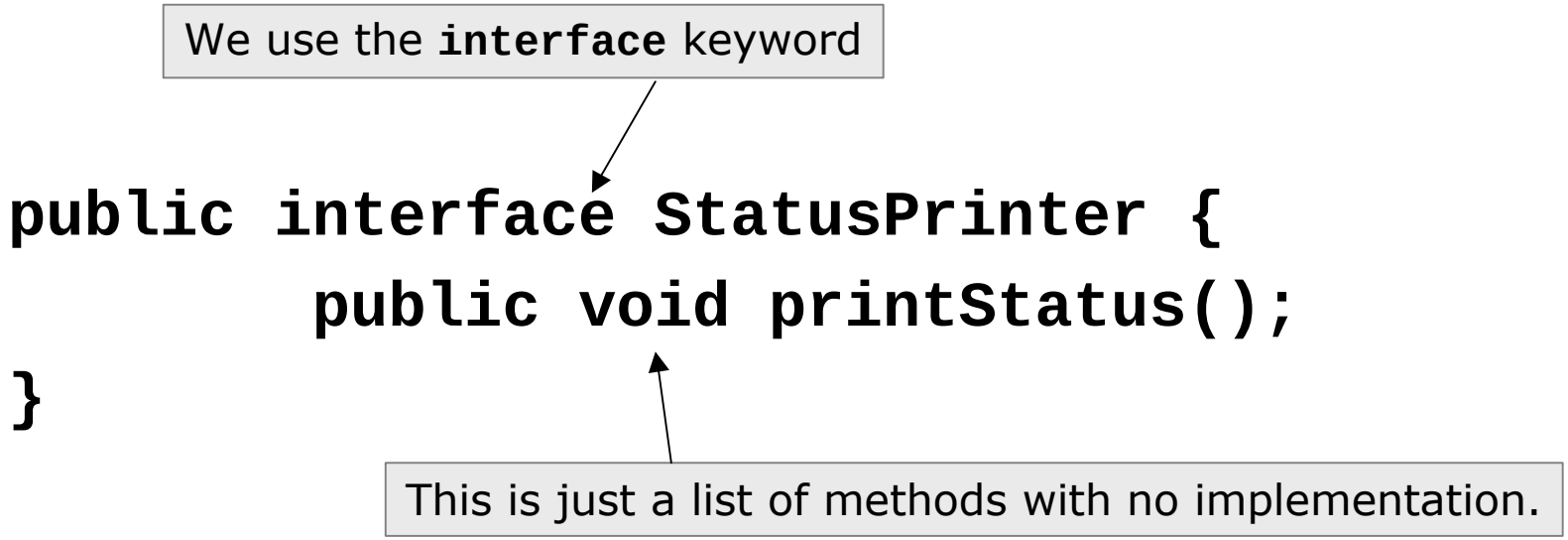
Interfaces

An interface is just a set of requirements we may ask a class definition to fulfill.

It is a list of methods and variables a class definition must implement.

In our case, we want every Counter and Accumulator to have a **`printStatus`** method.

Sce 5: StatusPrinter interface



This interface only includes one method. We could also define variables, use the public, protected and private attributes, specify lists of arguments, etc..

Sce 5: Accumulator class

```
public class Acumulator implements StatusPrinter{
        int sum = 0;
        public void add (int quantity) {
                sum = sum + quantity;
        }
        public int getSum () {
                return sum;
        }
        public void printStatus () {
             System.out.println ("I'm an accumulator \\
                        My sum is " + sum);
        }
}
```

We declare the class to follow the requirements specified in the StatusPrinter interface.

An we MUST provide an implementation for this method. Otherwise the compilation will fail.

Sce 5: Counter class

```
public class Counter implements StatusPrinter{
   .. .. ..
   public void printStatus() {
       System.out.println (“I’m a Counter. \\
           My value is “+value);
    }

   public int getValue() {
       return value;
   }
}
```

Sce 5: MyApplication class

```
public class MyApplication {

   static void displayStuff (StatusPrinter s) {
       s.printStatus();
   }

     public static void main (String args[]) {
        Accumulator a1 = new Accumulator();
        Counter     c1 = new Counter();
        MyApplication.displayStuff (a1);
        MyApplication.displayStuff (c1);
   }
}
```

A method which takes just a status printer.
No matter what the actual class of s is.

With interfaces we can make objects look the same in certain situations although they may belong to unrelated classes.
Interfaces are independent from the class hierarchy we build up.
Also we can overcome some advantages of not having multiple inheritance

SUMMARY

We have:

Seen what Java is.

Defined classes.
Instantiated objects from the classes defined.
Extended a class (creating an inheritance relation).
Redefined methods of a parent class.
Used object constructors.
Played around with static methods and variables.
Accessed objects with interfaces

SUMMARY

These are the building blocks with which to structure your code.

Building an OO application is to decide how to distribute and define access to your code by using classes, interfaces, exceptions, etc.

With AWT we will use the already made hierarchy of classes to manipulate graphical components. It s a good example to grasp how the OO conceptualization helps structuring the code.