The Java Series

Java Essentials
Advanced Language Constructs

Java Packages

- In OO, libraries contain mainly class definitions → A class hierarchy
- Typically, to use a class library we:
  - Instantiate objects from the classes in the library
  - Redefine the classes in the library (extending the class hierarchy).
  - Instantiate objects from the newly defined classes
- In Java class libraries are called **PACKAGES**.

Java Packages

- In Java there is no linking process.
- A **`.class`** file contains only code specific to the class it defines.
- A package is also a set of **`.class`** files.
- The compiler or interpreter dynamically load **`.class`** files as they are needed.
- **`.class`** files are looked for in a set of standard locations.
- This set can be extended with **`CLASSPATH`**

```
setenv CLASSPATH .:/opt/libs/CORBA:/opt/libs/simm
```

Java Packages

- Each package has a name:

  java.awt java.util java.io

- Whenever using packages we have to explicitly declare so. The following line at the beginning of a class definition:

```
import java.awt.*;
Window w = new Window();
```

  Means that we are going to use the classes defined in the java.awt package.

- In fact each class has a **fully qualified name** which includes the package it belongs to. So we can omit the import definition and write:

```
java.awt.Window w = new java.awt.Window();
```

Creating Java Packages

- Java packages are created by including:

  `package pckQualifier.pckname`

  at the beginning of every class in the package
- For instance:

  `package cern.it.utilities`
- When compiling packages, class files are generated in a directory structure resembling the package qualifier structure:

  `cern/`

  `cern/it`

  `cern/it/utilities`
- When using packages, the import statement will tell the interpreter where to look for starting off from what was specified in the CLASSPATH variable

Scenario 1: Creating a package

- We are going to create a package containing the Counter, BigCounter and Acccumulator classes from last tutorial. The package is going to be named: **`ch.cern.tutorials.javaEssentialsII`**
- Package naming convention to ensure uniqueness across the internet:
  - use your reversed DNS name as a preffix
- Package name structure MUST be parallel to the directory structure where class files will live.
- Tell javac to compile everything and create a directory structure to place compiled code:

```
javac -d . *.java
```

Package classes

```
package ch.cern.tutorials.javaEssentialsII;
public class Counter {
        int value = 0;
        public int increment() {
                value ++;
                return value;
        }
        public int getValue() {
                return value;
        }
}
```

```
package ch.cern.tutorials.javaEssentialsII;

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

```
package ch.cern.tutorials.javaEssentialsII;
public class BigCounter extends Counter{
        public int increment() {
                value = value + 2;
                return value;
        }
}
```

Sc1: User Application

```
import ch.cern.tutorials.javaEssentialsII.Counter;
import ch.cern.tutorials.javaEssentialsII.BigCounter;

public class MyApplication {

    public static void main (String args[]) {

        Counter c1 = new Counter();
        BigCounter c2 = new BigCounter();
        ch.cern.tutorials.javaEssentialsII.Accumulator
                a1 = new ch.cern.tutorials.javaEssentialsII.Accumulator();

         . . .

   }
}
```

Import classes from package and then we can use them directly.

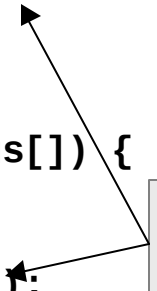
We don't import the class, so to access it we have to use the fully qualified name.

But to compile and run MyApplication I have to make the CLASSPATH variable point the location where the package ch.cern.tutorials.javaEssentialsII lives (never forget the current directory)

```
setenv CLASSPATH ../classes:.
```

Sc1: User Application

```
import ch.cern.tutorials.javaEssentialsII.*;

public class MyOtherApplication {

    public static void main (String args[]) {

        Counter c1 = new Counter();
        BigCounter c2 = new BigCounter();
        Accumulator a1 = new Accumulator();

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

        c1.increment();
        c1.increment();
        c2.increment();
        a1.add(30);

        System.out.println("Counter 1 now has value "+c1.getValue());
        System.out.println("Counter 2 now has value "+c2.getValue());
        System.out.println("Accumulator has value "+a1.getSum());

    }
}
```



Or we just import everything in the package and use it directly

jar files

- We can also ship packages in a single file containing all class definitions and directory structure.
- Java tools are able to understand directly shipped packages so that clients just have to worry about a single file.
- We use the jar jdk tool to create a single file for a package.
- Then we make the CLASSPATH point directly to the jar file

Creating and using jar files

```
[]> cd classes
[]> jar cvf javaEssentialsII.jar *

adding: ch/ (in=0) (out=0) (stored 0%)
adding: ch/cern/ (in=0) (out=0) (stored 0%)
adding: ch/cern/tutorials/ (in=0) (out=0) (stored 0%)
adding: ch/cern/tutorials/javaEssentialsII/ (in=0) (out=0) (stored 0%)
adding: ch/cern/tutorials/javaEssentialsII/Accumulator.class (in=429) (out=300) (deflated 30%)
adding: ch/cern/tutorials/javaEssentialsII/BigCounter.class (in=398) (out=272) (deflated 31%)
adding: ch/cern/tutorials/javaEssentialsII/Counter.class (in=428) (out=291) (deflated 32%)
```

We point directly the jar file

```
[]> setenv CLASSPATH ../javaEssentialsII.jar:.
[]> javac MyApplication
[]> javac MyOtherApplication
[]> java MyApplication
[]> java MyOtherApplication
```

More about jar files

- They are like zipped tar files containing other administrative information.
- java and javac unzip the jar files at run time.
- They are a convenient and compact way to deliver libraries.

- Very useful also for applets to save bandwidth and number of connections.

- They are also used to include certificates, encryption data, etc. about an package.

Primitive data types and references

- Java's primitive data types:
  - byte, short, int, long
  - float, double
  - char (16bit), boolean
- Apart from that everything is an object.
- Variables referring to objects are ALWAYS references to the actual object.
- When passing an object as a parameter it is ALWAYS passed as a reference.
- When passing an primitive data type as a parameter it is ALWAYS passed as a copy.

Sce 2: References and copies

```
class Test {
   int i=1;
   public void seti (int _i) { i=_i; }
   public int  geti ()       { return i; }
   public void incr ()       { i++; }
}

public class App {

  static void addToInteger (int i) { i++; }
  static void addToObject (Test t) { t.incr(); }

  public static void main (String args[]) {
    int j = 1;
    addToInteger(j);
    System.out.println("j's value is: "+j);

    Test t = new Test();
    addToObject(t);
    System.out.println ("t's integer is: "+t.geti());

  }
}
```

Here JVM makes a copy of I and then invokes the method

Here JVM just passes the reference, so manipulations within addToObject are on the very same object

The Garbage Collector

- For every object (instance) the JVM keeps a counter on how many references is has.
- Whenever the counter drops to 0, the JVM **AUTOMATICALLY** removes the instance from memory and it's not available anymore.
- This is called garbage collection and works recursively (if an instance containing the last reference to another object is removed this other object is also removed, etc…)
- Programmers don't have to worry explicitly about liberating memory space of objects they created. This implies that there is no idea of destructor as in C++.
- You can write a finalize() method in any class, and the garbage collector will call it just before removing any instance of that class.
- You can call explicitly the garbage collector: **`Runtime.gc();`**

Control Flow Statements

- Of course Java contains control flow statements to use in your methods:

```
 if ( myObj.getID == 0 ) {
    System.out.println(“myObject is 0”);
else {
   System.out.pritln(“myObject is not 0”);
}

while (myCounter.getValue()<10) {
    myCounter.increment();
}

for (int i=0; i<10; i++) {
     some code
}
```

- And there are more: switch, case, elseif, etc..

Strings and arrays

- You can also create arrays of objects:

```
Counter counters[] = new Counter[1000];
```

- And then access each counter:

```
counters[i] = new Counter();
```

- Creating an array doesn't create the instances.
- Java automatically checks for arrays limits and generates runtime errors if you exceed them.

- A String is just a regular object containing an array of chars with a bunch of methods to manipulate it.
- Java also provides the "" syntax to implicitly deal with String objects in a familiar way
- Strings are immutable, different "" originate different objects (instances). Garbage collector does the rest.

Sce 3: Arrays

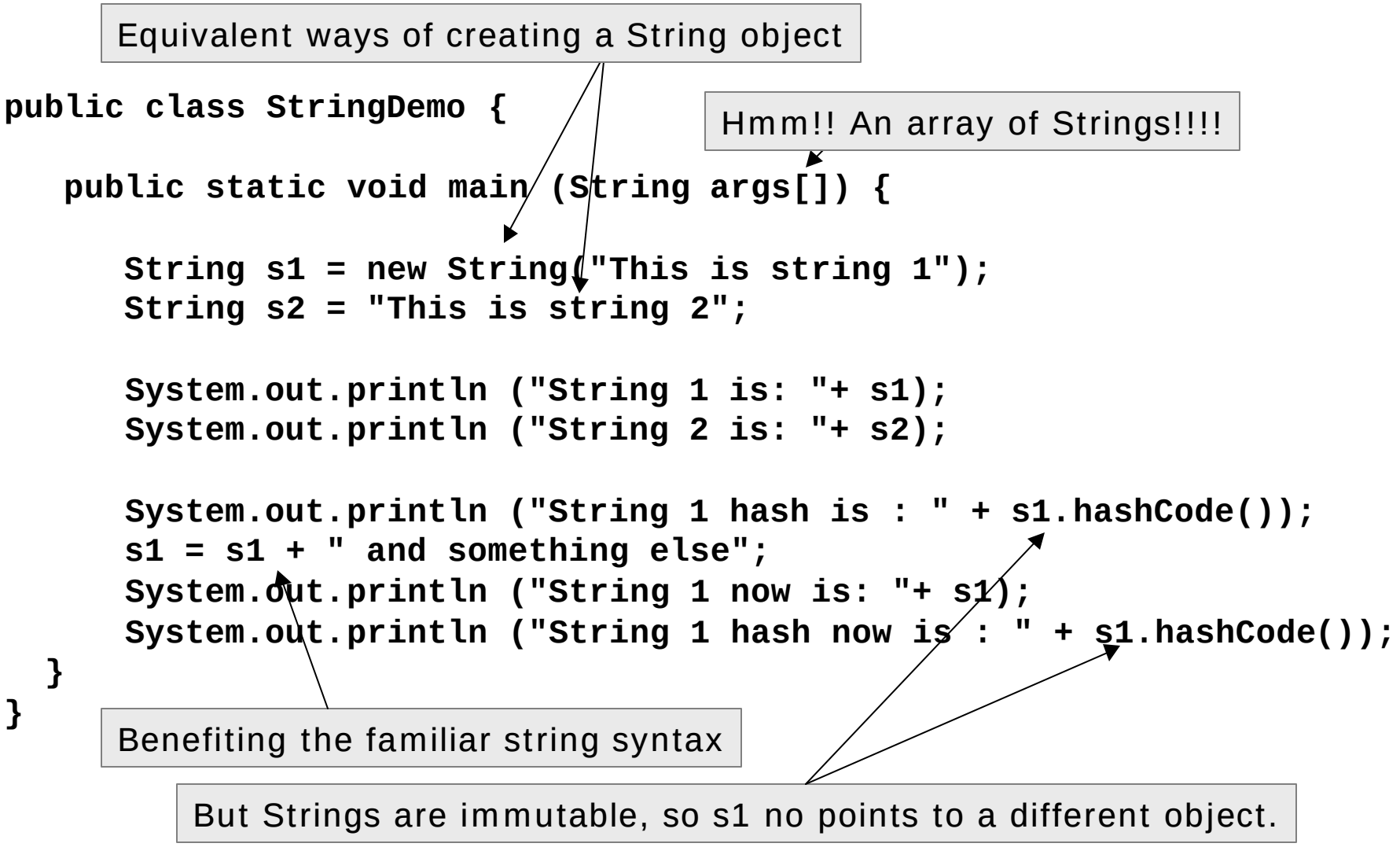
```
public class ArrayDemo {
  public static void main(String[] args) {

    Counter[] counters;
    counters = new Counter[10];
    for (int i=0; i<10; i++) {
        counters[i] = new Counter();
    }

    System.out.println("Array length is " + counters.length);
    for (int i=0; i<12; i++) {
        System.out.println("Counter "+i+" value is:"
            + counters[i].getValue());
    }
  }
}
```

The array declaration and creation are separated

We create the instances to fill in the array

We try to access over the array bounds

Sce3: Strings




```
public class StringDemo {

   public static void main (String args[]) {

      String s1 = new String("This is string 1");
      String s2 = "This is string 2";

      System.out.println ("String 1 is: "+ s1);
      System.out.println ("String 2 is: "+ s2);

      System.out.println ("String 1 hash is : " + s1.hashCode());
      s1 = s1 + " and something else";
      System.out.println ("String 1 now is: "+ s1);
      System.out.println ("String 1 hash now is : " + s1.hashCode());
   }
}
```

jdk packages

- jdk comes with a set of useful packages:
  - java.lang: Basic language definitions
  - java.io: I/O classes
  - java.util: General purpose classes.
  - java.awt: GUI building
  - java.swing: Advance GUI building
  - java.sql: Database access
  - java.rmi: Remote objects
- we are going to look at a couple of useful classes in the java.util package.

Vectors

- A vector is a growable array of objects
- It contains objects to access through an index.
- The size can grow or shrink as elements are added or removed.
- You have certain control on how space is allocated and deallocated:

```
Vector v = new Vector()          // Initial size is 10
Vector v = new Vector(1000);
Vector v = new Vector(1000,15);
```

- And you have methods to add and remove objects
  - v.add (new Counter());
  - v.add (c1);
- Keep in mind that, in theory, you can have different kinds of objects in the same vector.
- To avoid complications a vector should contain objects of the same class.

Sce4: Vectors

```
import java.util.*;

public class VectorDemo {
   public static void main (String args[]) {

      Vector v = new Vector();
      v.addElement(new Counter(14));
      v.addElement("This is a string");

      Counter c = (Counter)v.elementAt(0);
      c.increment();
      System.out.println("Counter is : " + c.getValue());

      String s = (String)v.elementAt(1);
      System.out.println ("String is " + s);
      System.out.println ("String hash code is " + s.hashCode());

      v.removeElementAt(0);
      String t = (String)v.elementAt(0);
      System.out.println ("String is " + t);
      System.out.println ("String hash code is " + t.hashCode());

  }
}
```

Create a vector and add elements

When retrieving elements we have to “cast” them

When removing an element, the following ones are shifted

See java.util.Vector api!!!

Hash tables

- Hash tables map keys to elements. They are a generalization of vectors (map indexes to elements)
  - "John" -> Counter c1
  - "Don" -> Counter c2
  - …
- Values are then retrieved by specifying the key.

```
Hashtable h = new Hashtable();
h.put("John", new Counter());
Counter c = (Counter)h.get("John");
```

- Either keys or elements can be any object. It's very general
- Hastables can be iterated:
  - We can iterate through the set of keys
  - We can iterate through the values
- Iteration is done through the Enumeration class.

```
Enumeration e = h.getElements();
Enumeration e = h.getKeys();
```

Sc4: HashTables

```
import java.util.*;

public class HashTableDemo {

   public static void main (String args[]) {

     Hashtable h = new Hashtable();
     h.put ("one", new Counter(10));
     h.put ("two", new Counter(4));
     h.put ("three", new Counter(2));

     Counter c = (Counter)h.get("three");
     System.out.println (”three is: "+c.getValue());
     h.remove("three");

     for (Enumeration e = h.keys(); e.hasMoreElements(); ) {
          String key = (String)e.nextElement();
          Counter cc = (Counter)h.get(key);
          System.out.println ("Counter "+key+" value is: "+cc.getValue());
     }

     for (Enumeration e = h.elements(); e.hasMoreElements(); ) {
          Counter cc = (Counter)e.nextElement();
          System.out.println ("Counter's value is: "+cc.getValue());
     }
  }
}
```

Create hash table and associate elements

Retrieve an element (we have to cast it)

Remove an element

Iterate through keys and then access elements

Iterate directly through elements.

Sample Scenario 5

- Now we want every counter to have a top value so that when ask to increment above that value an error is produced.

Error Handling

- Typically errors are implemented as an special return value (-1) that the caller must not forget to check. Two problems:
  - We may not have free return values to use as error codes (i.e. division). We are overloading the return value.
  - We cannot force the caller to check the error.

Exceptions

- We are really needing two channels when invoking methods:
  - One for the return value
  - One for the error code (if any)
- In Java this is provided with EXCEPTIONS
  - An EXCEPTION is just another object itself.
  - The base class is the Exception class.

How Exceptions work

- The method where the error may occur must
  - 1. Create an exception object (with the appropriate data within -error code,...-)
  - 2. Send the exception through the “error channel”
  - 3. Declare it may be using the “error channel”
- The caller must:
  - 1. Provide the code to receive the error in case any is sent through the channel
- In Java, the terminology is:
  - To **throw** an exception (called).
  - To **catch** an exception (caller).

Exception handling

- Now, the compiler knows what methods may be using the errors channel, and thus, may be **throwing exceptions**.
- So the compiler **forces** any caller using those methods to include code to **catch those exceptions**.
- If you forget to include that code, then the compiler will produce an error.

Sce 5: Counter class

```
public class Counter {
        int value = 0;
        int topValue = 5;
        public int increment() throws Exception {
                if (value >= topValue)
                        throw (new Exception());
                value ++;
                return value;
        }
        public int getValue() {
                return value;
        }
}
```

The method declares it may throw an exception

So, when we are interested in using the exceptions channel we just:

1. Create an exception object
2. Throw it
3. Java takes care of passing it on to the caller

Sce 5: MyApplication class

```
public class MyApplication {

    public static void main (String args[]) {

        Counter c1 = new Counter();

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

        try {
           for (int i=1; i<10; i++) {
               c1.increment();
               System.out.println("Counter 1 is now "+c1.getValue());
           }
        } catch (Exception e) {
           System.out.println("I just caught the Exception");
        }

    }
}
```

This piece of code will be executed until it finishes or an exception is thrown

This piece of code will be executed if an exception is thrown while executing the **`try`** section.

Sce 5: If we forget it

If we forget the **`try - catch`** block:

```
rsplus03> javac MyApplication.java

MyApplication.java:11: Exception java.lang.Exception must be
   caught, or it must be declared in the throws clause of this
   method.
                c1.increment();
                               ^
1 error
```

But Remember

- Exceptions are object just as any other object in our application.
- So we can create or own exceptions, by redefining the already existing **`Exception`** class.
- So in the following scenario we are going to create our own exceptions.

Scenario 6: CounterException class

Inherits everything from the built-in Exception class

```
public class CounterException extends Exception {
        String message;
        public CounterException (String msg) {
                message = msg;
        }
        public String getErrorMessage() {
                return message;
        }
}
```

We specify a constructor so that whoever creates a CounterException also specifies some error message.

Sce 6: Counter class

```
public class Counter {
   int value = 0;
   int topValue = 5;
   public int increment() throws CounterException {
       if (value >= topValue)
         throw (new CounterException("Counter exceeded"));
       value ++;
       return value;
   }
   public int getValue() {
       return value;
   }
}
```

We declare it throws a CounterException.

We create a CounterException and specify an error message.

Sce 6: MyApplication class

```
public class MyApplication {

    public static void main (String args[]) {

        Counter c1 = new Counter();

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

        try {
           for (int i=1; i<10; i++) {
               c1.increment();
               System.out.println("Counter 1 is now "+c1.getValue());
           }
        } catch (CounterException e) {
           System.out.println(e.getErrorMessage());
        }

    }
}
```

We catch a CounterException.

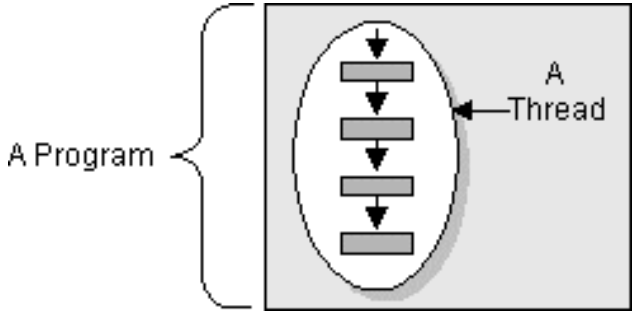
Once we have it we deal with it as with any other object.

Methods' Signature

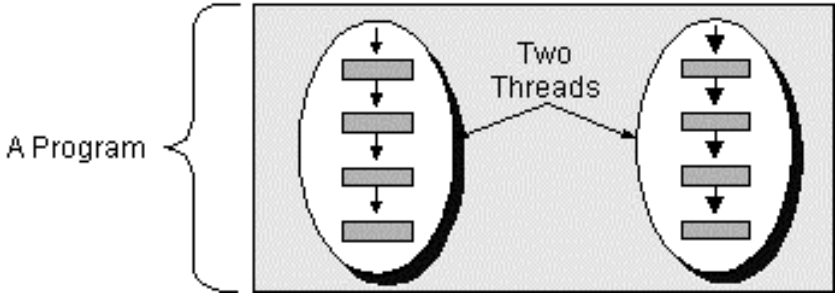
- One may consider that a method signature is now composed of:
  - Method name
  - Return value
  - Number and type of arguments
  - Number and type of Exceptions thrown
- And with all the consequences in derived classes.

Threads

- A Thread is a sequential flow of control within a program:



- There can be many threads (ex. browsers, etc.)



- And we control its creation, destruction, etc.

SC7: Implementing Threads

- One way: Subclassing the Thread class

```
public class SimpleThread extends Thread {
   public SimpleThread(String str) {
       super(str);
   }
   public void run() {
        for (int i = 0; i < 10; i++) {
           System.out.println(i + " " + getName());
           try {
               sleep((long)(Math.random() * 1000));
           } catch (InterruptedException e) {}
       }
       System.out.println("DONE! " + getName());
   }
}
```

Implement the run method

```
public class MyApplication {
   public static void main (String[] args) {
       new SimpleThread("Jamaica").start();
       new SimpleThread("Fiji").start();
   }}
```

Use the start method

SC8: Implementing Threads

- Another way: Implementing the Runnable interface

```
public class DoTen implements Runnable {

   Thread myThread = null;
   String myName = "";
   public DoTen (String str) {
      myName = str;
   }
   public void startCounting() {
      if (myThread == null) {
        myThread = new Thread (this, myName);
        myThread.start();
      }
   }
   public void run() {
      for (int i = 0; i < 10; i++) {
          System.out.println(i + " " + myThread.getName());
          try {
             myThread.sleep((long)(Math.random() * 1000));
          } catch (InterruptedException e) {}
      }
      System.out.println("DONE! " + myThread.getName());
   }
}
```

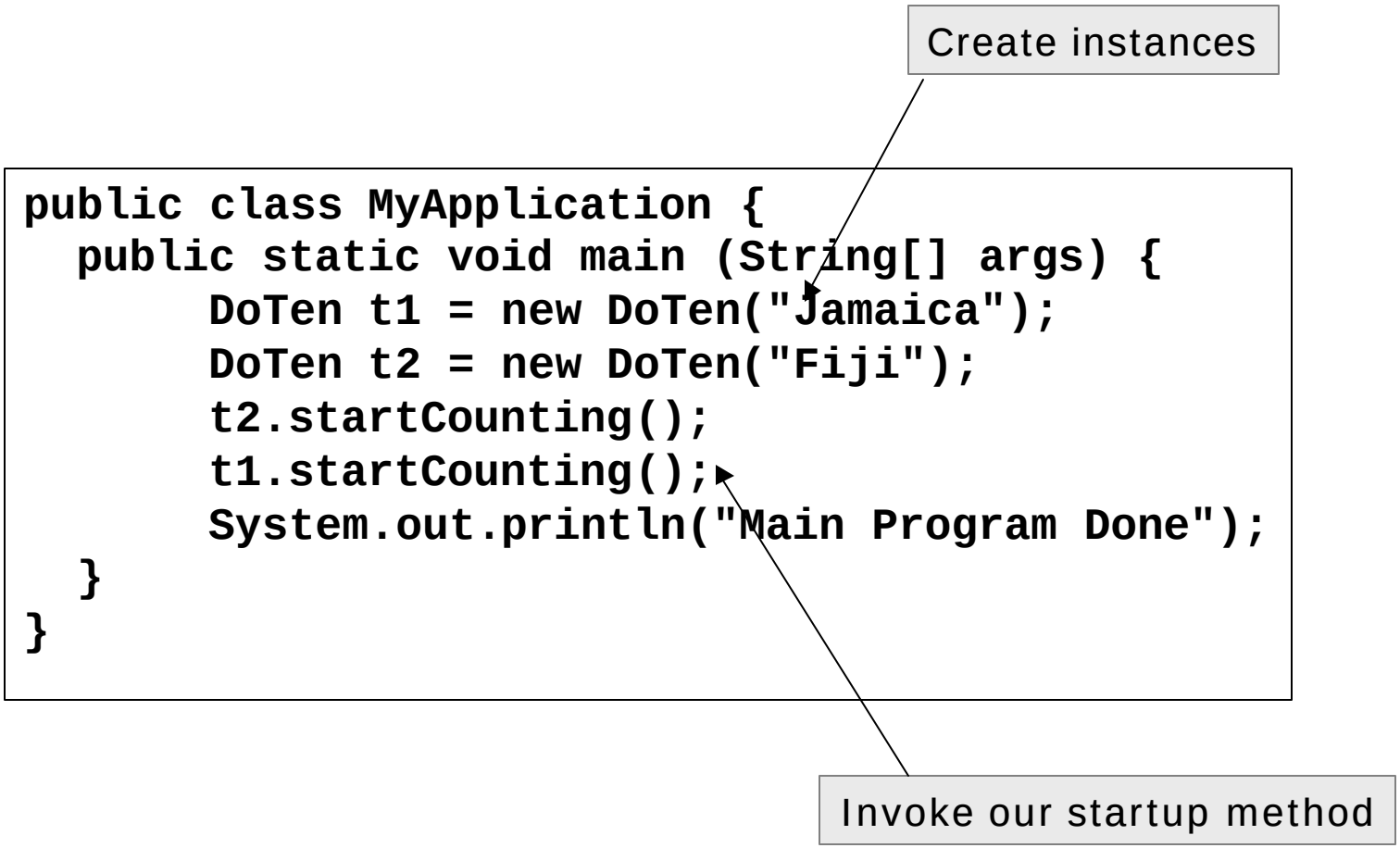
Must have own thread

We manipulate the thread. Our decision

Implement the run method

SC8: Implementing Threads

- Another way: Implementing the Runnable interface



More on Threads

- Use Runnable interface if you need to respect your class hierarchy (this is what interfaces are for!!!)
- Can set Threads **priorities** to share CPU:

```
Thread t = new Thread("test")
t.setPriority (2);
```

- Threads with higher priorities get CPU time.
- Threads with the same priority in round-robin.
- For example: no priority 1 thread will get CPU until all priority 2 threads are done.
- Methods (in any object) can be **synchronized**

```
public class myClass {
    public sychronized method Calculate() { …… }
}
```

- Java takes care no two threads run at the same time synchronized methods of any object.

Summary

- This is a first appetizer of what Java programming is like.
- We have seen how to deploy and use packages.
- We know about
  - The garbage collector
  - Parameters & references
  - Control structures
- We have seen some useful classes :
  - Arrays, Strings
  - Vector, Hashtables
- We have seen how to handle and build an architecture of exceptions.
- And played around with Threads