PROGRAMMING

Lecture 23

Sushil Paudel

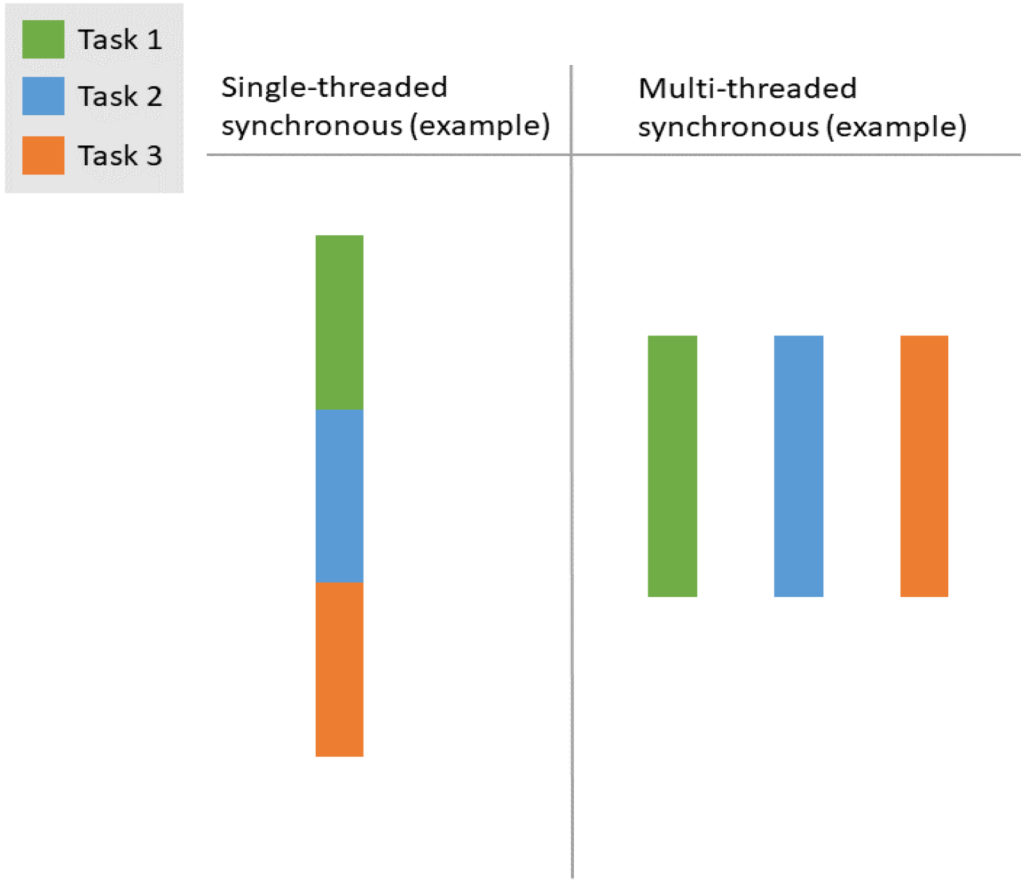
TODAY'S TOPIC

- Multithreading
- Java Date
- Final keyword
- Programming standards

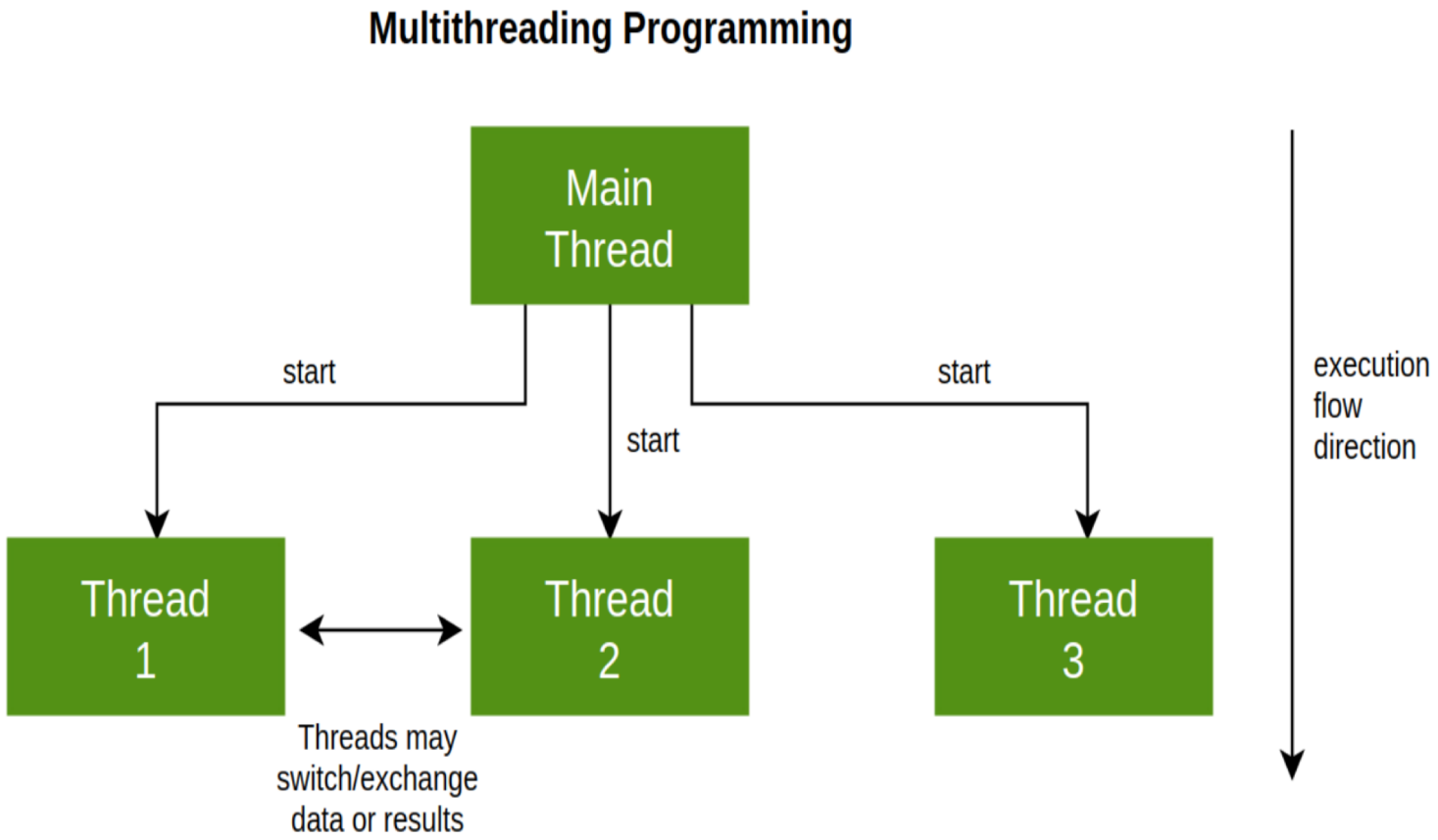
MULTITHREADING

- Multithreading in Java is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing.
- A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time to maximum utilization of CPU.

MULTITHREADING



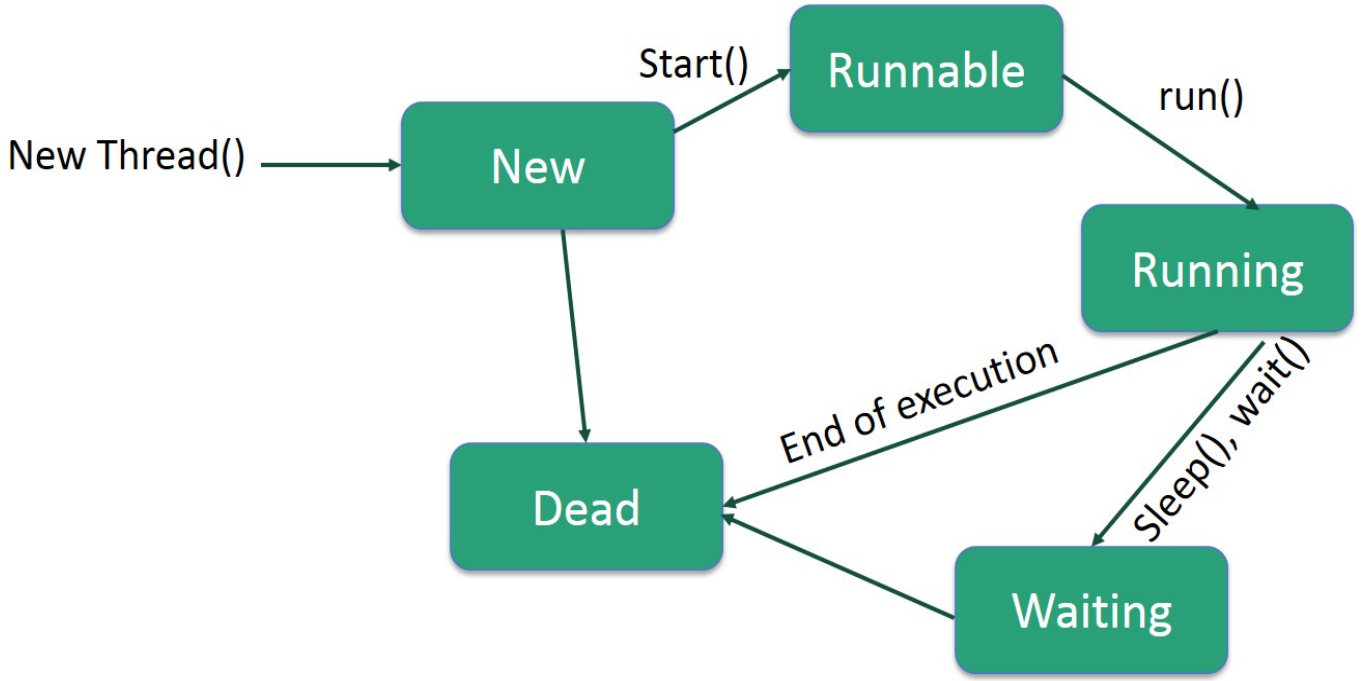
MULTITHREADING



ADVANTAGE OF MULTITHREADING

- It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
- You **can perform many operations together, so it saves time**.
- Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

LIFE CYCLE OF A THREAD



EXAMPLE

```
public class MultithreadingDemo extends Thread {

    public void run() {
        System.out.println("Running " + Thread.currentThread().getId());
        try {
            for (int i = 0; i < 2; i++) {
                System.out.println(Thread.currentThread().getId() + ": " + i);
                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + Thread.currentThread().getId() + " interrupted.");
        }
        System.out.println("Thread " + Thread.currentThread().getId()+ " exiting.");
    }
}
```

EXAMPLE

```
public class MultithreadTest {

    public static void main(String[] args) {
        for (int i = 0; i < 3; i++) {
            MultithreadingDemo object = new MultithreadingDemo();
            object.start();
        }
    }
}
```

OUTPUT

```
Running 10
Running 12
Running 11
12: 0
10: 0
11: 0
10: 1
11: 1
12: 1
Thread 10 exiting.
Thread 12 exiting.
Thread 11 exiting.
```

EXAMPLE WITHOUT THREAD

```
public class MultithreadingDemo {
    public void run() {
        System.out.println("Running " + Thread.currentThread().getId());
        try {
            for (int i = 0; i < 2; i++) {
                System.out.println(Thread.currentThread().getId() + ": " + i);
                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + Thread.currentThread().getId() + " interrupted.");
        }
        System.out.println("Thread " + Thread.currentThread().getId()+ " exiting.");
    }
}
```

EXAMPLE WITHOUT THREAD

```
public class MultithreadTest {

    public static void main(String[] args) {
        for (int i = 0; i < 3; i++) {
            MultithreadingDemo object = new MultithreadingDemo();
            object.run();
        }
    }
}
```

OUTPUT

```
Running 1
1: 0
1: 1
Thread 1 exiting.
Running 1
1: 0
1: 1
Thread 1 exiting.
Running 1
1: 0
1: 1
Thread 1 exiting.
```

JAVA CALENDAR

```
public class MyCalendar {

	public static void main(String args[]) {
		Calendar c = Calendar.getInstance();
		System.out.println("The Current Date is:" + c.getTime());
	}
}
```

Output

The Current Date is:Tue Sep 01 08:31:10 NPT 2020

JAVA DATE

```
public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();
        System.out.println(date.toString());
    }
}
```

Output

Tue Sep 01 08:31:59 NPT 2020

JAVA DATE DIFFERENCE

```
public class DateDifference {

    public static long getDateDiff(Date date1, Date date2) {
        long diffInMillies = date2.getTime() - date1.getTime();
        return diffInMillies;
    }
}
```

Output

Difference: 1015

FINAL KEYWORD

- In Java, the final keyword is used to denote constants.
- It can be used with variables, methods, and classes.

- Once any entity (variable, method or class) is declared final, it can be assigned only once. That is,
  - the final variable cannot be reinitialized with another value
  - the final method cannot be overridden
  - the final class cannot be extended

FINAL VARIABLE

In Java, we cannot change the value of a final variable. For example,

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

    // create a final variable
    final int AGE = 32;

    // try to change the final variable
    AGE = 45;
    System.out.println("Age: " + AGE);
  }
}
```

FINAL VARIABLE

- In the above program, we have created a final variable named *age*. And we have tried to change the value of the final variable.

- When we run the program, we will get a compilation error with the following message.

```
cannot assign a value to final variable AGE
    AGE = 45;
    ^
```

FINAL METHOD

In Java, the final method cannot be overridden by the child class. For example,

```
class FinalDemo {
    // create a final method
    public final void display() {
      System.out.println("This is a final method.");
    }
}

class Main extends FinalDemo {
  // try to override final method
  public final void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

FINAL METHOD

- In the above example, we have created a final method named display() inside the FinalDemo class.
- Here, the *Main* class inherits the *FinalDemo* class.
- We have tried to override the final method in the *Main* class. When we run the program, we will get a compilation error with the following message.

```
display() in Main cannot override display() in FinalDemo
 public final void display() {
                   ^
 overridden method is final
```

FINAL CLASS

- In Java, the final class cannot be inherited by another class. For example,

```
// create a final class
final class FinalClass {
  public void display() {
    System.out.println("This is a final method.");
  }
}

// try to extend the final class
class Main extends FinalClass {
  public  void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

FINAL CLASS

- In the above example, we have created a final class named *FinalClass*. Here, we have tried to inherit the final class by the *Main* class.
- When we run the program, we will get a compilation error with the following message.

```
cannot inherit from final FinalClass
class Main extends FinalClass {
                   ^
```

COMPILING PACKAGE FROM COMMAND

Suppose the file structure is as below:

- src
  - week20
    - Student
    - FirstYear
    - FinalYear
    - StudentEntry

1. Go to Command line and make sure you are in *src* folder.
2. Enter the command to compile: *javac week20/StudentEntry.java*
3. Enter the command to run: *java week20/StudentEntry*

PROGRAMMING STANDARDS

- Use meaningful names.

  Use descriptive names for all identifiers (names of classes, variables and methods). Avoid ambiguity. Avoid abbreviations. Simple mutator methods should be named set*Something*(...). Simple accessor methods should be named get*Something*(...). Accessor methods with boolean return values are often called is*Something*(...), for example, isEmpty().

- Class names start with a capital letter.
- Class names are singular nouns.
- Method and variable names start with lowercase letters.

PROGRAMMING STANDARDS

- All three - class, method and variable names - use capital letters in the middle to increase readability of compound identifiers, e.g. numberOfItems.
- Constants are written in UPPERCASE.
  - Constants occasionally use underscores to indicate compound identifiers: MAXIMUM_SIZE

PROGRAMMING STANDARDS

- One level of indentation is four spaces.
- All statements within a block are indented one level.
- Use a space before the opening brace of a control structure's block.
- Use a space around operators.
- Use a blank line between methods (and constructors).
    - Use blank lines to separate logical blocks of code. This means at least between methods, but also between logical parts within a method.

PROGRAMMING STANDARDS

- Every class has a class comment at the top.
- The class comment contains at least
  - a general description of the class
  - the author's name(s)
  - a version number
- Every person who has contributed to the class has to be named as an author or has to be otherwise appropriately credited.

PROGRAMMING STANDARDS

- Every method has a method comment.
- Class and method comments must be recognised by Javadoc. In other words: they should start with the comment symbol /**.
- Code comments (only) where necessary.
- Comments in the code should be included where the code is not obvious or difficult to understand (while preference should be given to make the code obvious or easy to understand where possible), and where it helps understanding of a method.

PROGRAMMING STANDARDS

- Order of declarations: fields, constructors, methods.
  - The elements of a class definition appear (if present) in the following order: package statement; import statements; class comment; class header; field definitions; constructors; methods.
- Fields may not be public (except for final fields).
- Always use an access modifier.
  - Specify all fields and methods as either private, public, or protected. Never use default (package private) access.

PROGRAMMING STANDARDS

- Import classes separately.
  - Import statements explicitly naming every class are preferred over importing whole packages. E.g.

    import java.util.ArrayList;

    import java.util.HashSet;

    is better than

    import java.util.*;

- Initialize all the required fields in the constructor.

THANK YOU!

Any questions?