Interfaces

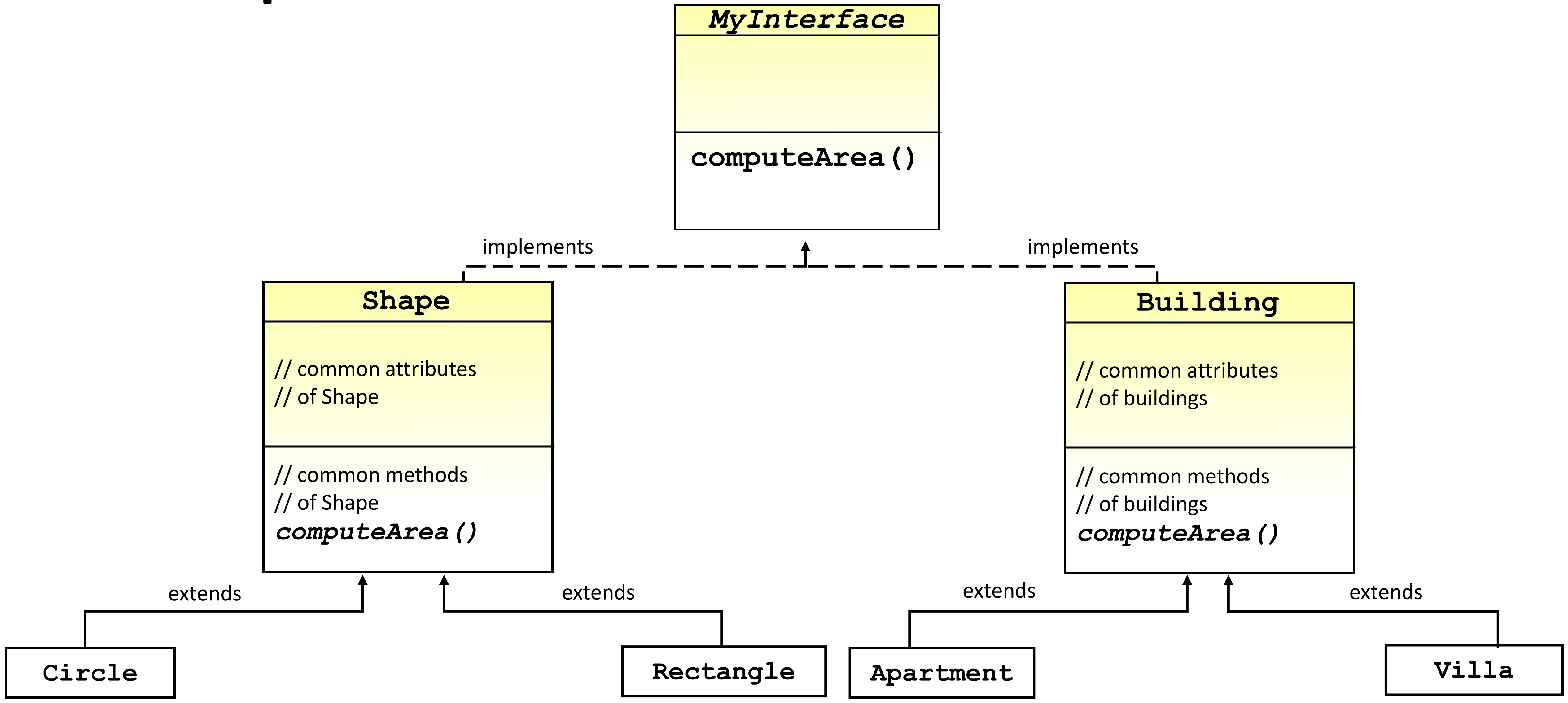
Interfaces

- An interface defines a standard & public way of specifying the behavior of classes:
  - It is said to be a **contract between two classes**.
- An interface is like a fully abstract class:
  - All of its methods are abstract.
  - All variables are `public static final`.

Interfaces

- An interface allows classes, regardless of their locations in the class hierarchy, to implement common behaviors.
  - It is a means for defining common behavior using method signatures without any body.
    - It does not state what a method does; it just defines how it should look like.
- An interface may be regarded as a workaround solution for the problems of multiple inheritance.

Example of Interfaces



Interfaces cont'd

- The following is the code for declaring **`MyInterface`**:

```
public interface MyInterface
{
        float PI = 3.14f ;

        float computeArea();
}
```

- An interface is written inside a normal **`.java`** file.
- Methods of an interface are by default:

  `public` and **`abstract`**.
- Variables are by default:

  `public`, **`static`**, and **`final`**.

Interfaces cont’d

- When a class chooses to implement an interface,
  - it is obligatory that the class overrides (implements) all the methods of that interface.
  - If even one method is left unimplemented, then
    - the compiler requires that you declare your class as an abstract class.

```
public class MyClass implements MyInterface{
        public float computeArea(){
                //calculate and return area here
        }

        public void anotherMethod(){
                //you are allowed to have any other
                //methods in the class
        }
}
```

Interfaces cont'd

- A reference of an interface can refer to an object of a class that implements that interface
  - just as same as the relation between a parent class and its child:

```
MyInterface mi = new MyClass();

mi.computeArea();

mi.anotherMethod();  //ILLEGAL
```

- Such feature is useful,
  so that when a certain method requires an Interface type as one of its parameters,
  you can, flexibly enough, pass an object of any class that has implemented that interface.

Interfaces

Here is an example of an **interface** definition.

```
public interface Numbers
{
  int getNext(); // return next number in series
  void reset(); // restart
  void setStart(int x); // set starting value
}
```

Implementing Interfaces

```
public interface Numbers {
    int getNext();
    void reset();
    void setStart(int x);
}
```

```
class ByTwos implements Numbers{
    int start;
    int val;
    public ByTwos() {
        start = 0;
        val = 0;
     }
    public int getNext() {
        val += 2;
        return val;
    }
    public void reset() {
        val = start;
    }
    public void setStart(int x){
        start = x;
        val = x;
    }
}
```

Note that the methods getNext( ), reset( ), and setStart( ) are declared using the public access specifier

```
public class Demo {
  public static void main (String args[]) {
        ByTwos ob = new ByTwos();
        for (int i = 0; i < 5; i++) {
                System.out.println("Next value is " + ob.getNext());
        System.out.println("\n Resetting");
        ob.reset();
        for (int i = 0; i < 5; i++)
                System.out.println("Next value is " +ob.getNext());
        System.out.println("\n Starting at 100");
        ob.setStart(100);
        for (int i = 0; i < 5; i++)
                System.out.println("Next value is " + ob.getNext());
  }
}
```

Implementing Interfaces

```
public interface Numbers {
    int getNext();
    void reset();
    void setStart(int x);
}
```

```
class ByThrees implements Numbers {

    int start;
    int val;
    public ByThrees() {
        start = 0;
        val = 0;
    }
    public int getNext() {
        val += 3;
        return val;
    }
    public void reset() {
        val = start;
    }
    public void setStart(int x) {
        start = x;
        val = x;
    }
}
```

- Class **ByThrees** provides another implementation of the **Numbers** interface
- Notice that the methods getNext( ), reset( ), and setStart( ) are declared using the public access specifier

Implementing Interfaces

```
class Demo2 {
   public static void main (String args[]) {
        ByTwos twoOb = new ByTwos();
        ByThrees threeOb = new ByThrees();
        Numbers ob;
        for(int i=0; i < 5; i++) {
                ob = twoOb;
                System.out.println("Next ByTwos value is " + ob.getNext());
                ob = threeOb;
                System.out.println("Next ByThrees value is " + ob.getNext());
        }
   }
}
```

Multi-Threading

What is Thread?

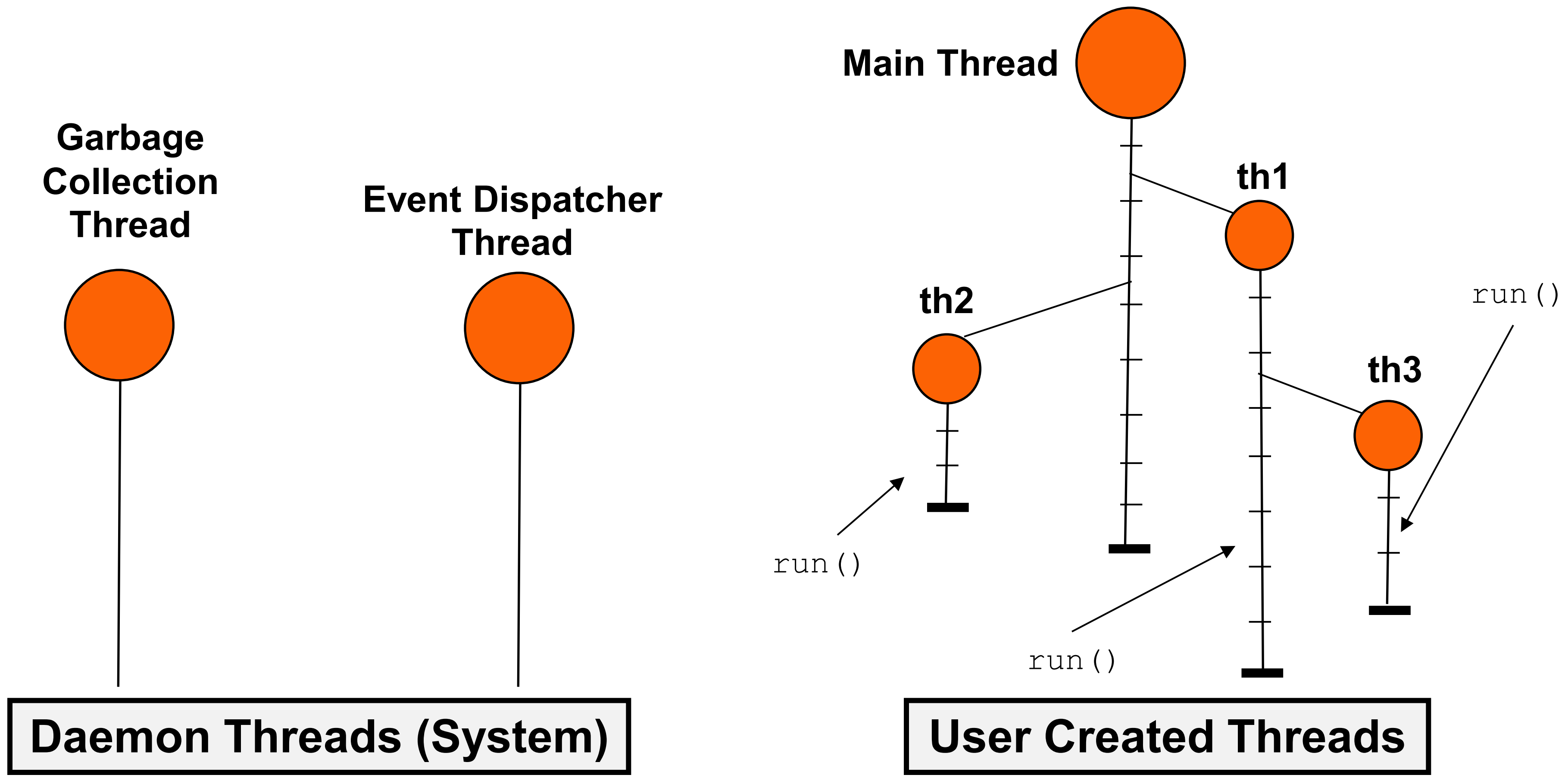
- A Thread is
  - A single sequential execution path in a program
  - Used when we need to execute two or more program segments concurrently (multithreading).
  - Used in many applications:
    - Games, animation, perform I/O
  - Every program has at least two threads.
  - Each thread has its own stack, priority & virtual set of registers.

What is Thread?

- Multiple threads do not mean that they execute in parallel when you’re working in a single CPU.
  - Some kind of scheduling algorithm is used to manage the threads (e.g. Round Robin).
  - The scheduling algorithm is JVM specific (i.e. depending on the scheduling algorithm of the underlying operating system)

Threads

- several thread objects that are executing concurrently:



Threads cont'd

- Threads that are ready for execution are put in the ready queue.
  - Only one thread is executing at a time, while the others are waiting for their turn.
- The task that the thread carries out is written inside the `run()` method.

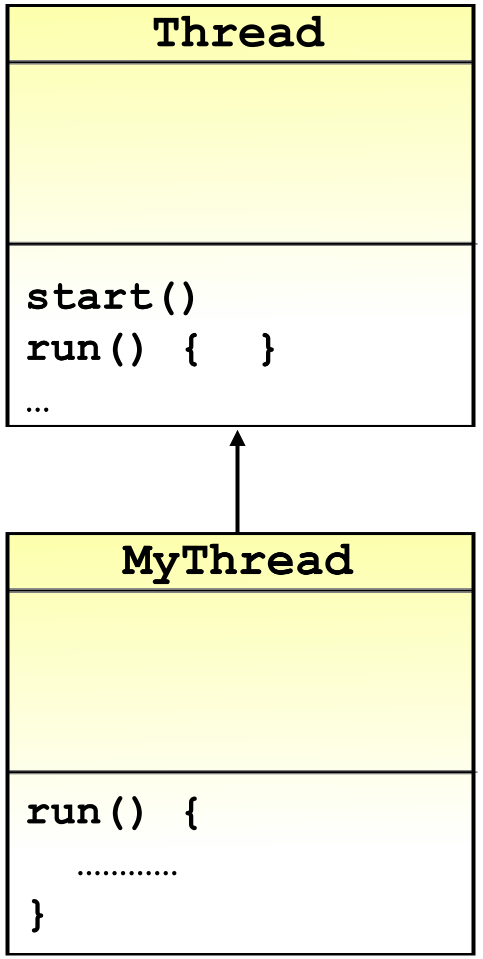
The `Thread` Class

- `Class Thread`
  - `start()`
  - `run()`
  - `sleep()`
  - `suspend()`*
  - `resume()`*
  - `stop()`*
- `Class Object`
  - `wait()`
  - `notify()`
  - `notifyAll()`

* *Deprecated Methods (may cause deadlocks in some situations)*

Working with Threads

- There are two ways to work with threads:
  - Extending Class **`Thread`**:
    1. Define a class that extends **`Thread`**.
    2. Override its **`run()`** method.
    3. In main or any other method:
       a. Create an object of the subclass.
       b. Call method **`start()`**.

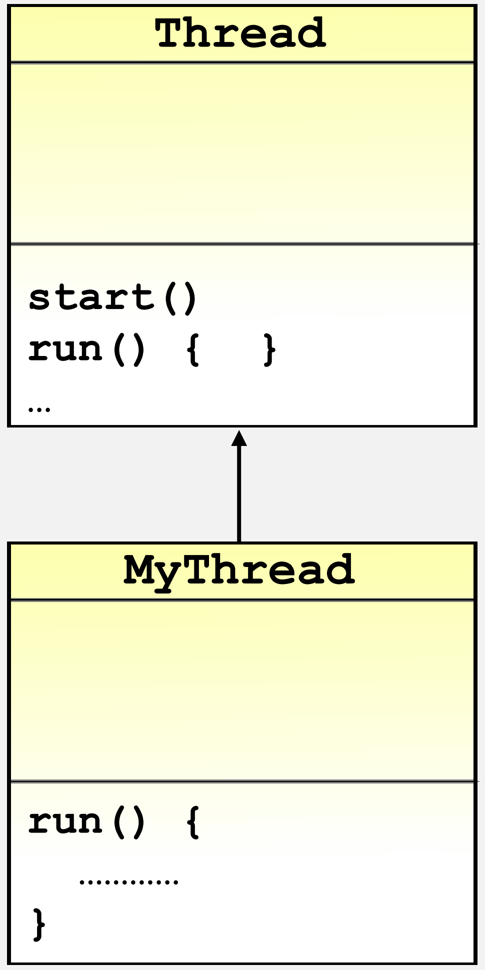


Working with Threads cont'd

```
public class MyThread extends Thread (1)
{
   public void run( ) (2)
   {
        … //write the job here
   }
}
```

- in **main()** or any method:

```
public void anyMethod()
{
   MyThread th = new MyThread(); (3.a)
   th.start(); (3.b)
}
```



Working with Threads

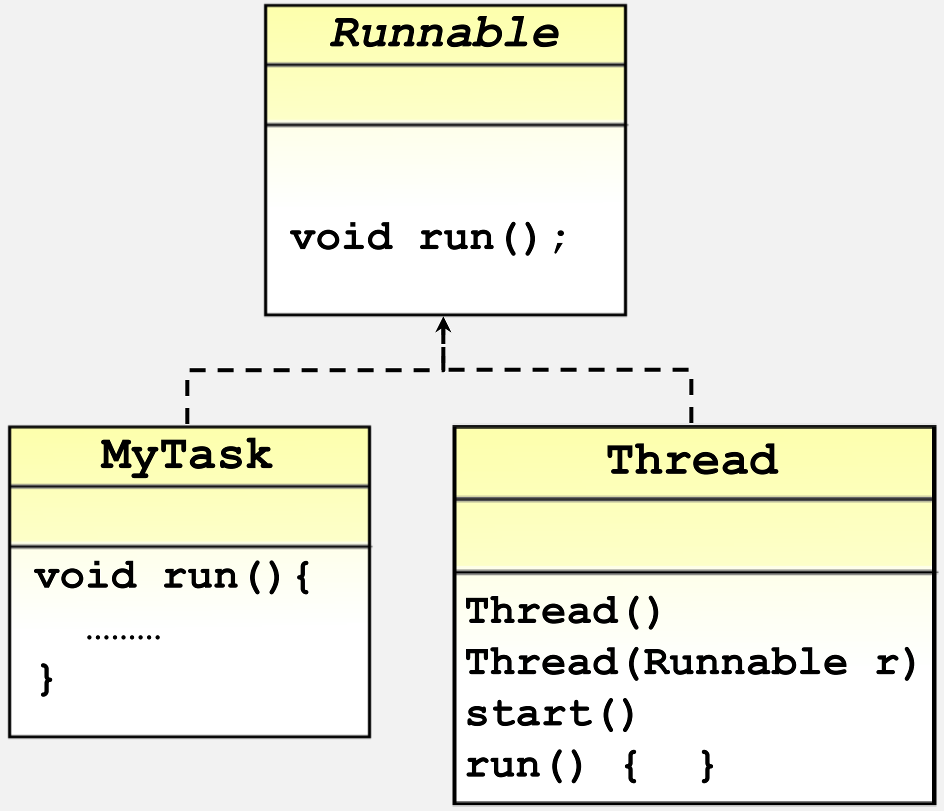
- There are two ways to work with threads:
  - Implementing Interface ***`Runnable`***:
    1. Define a class that implements **`Runnable`**.
    2. Override its **`run()`** method .
    3. In main or any other method:
        a. Create an object of your class.
        b. Create an object of class **`Thread`** by passing your object to the constructor that requires a parameter of type **`Runnable`**.
        c. Call method **`start()`** on the **`Thread`** object.

Working with Threads cont'd

```
class MyTask implements Runnable  1
{
   public void run()  2
   {
        … //write the job here
   }
}
```

- in **main()** or any method:

3
```
public void anyMethod()
{
   MyTask task = new MyTask();  a
   Thread th = new Thread(task);  b
   th.start();  c
}
```

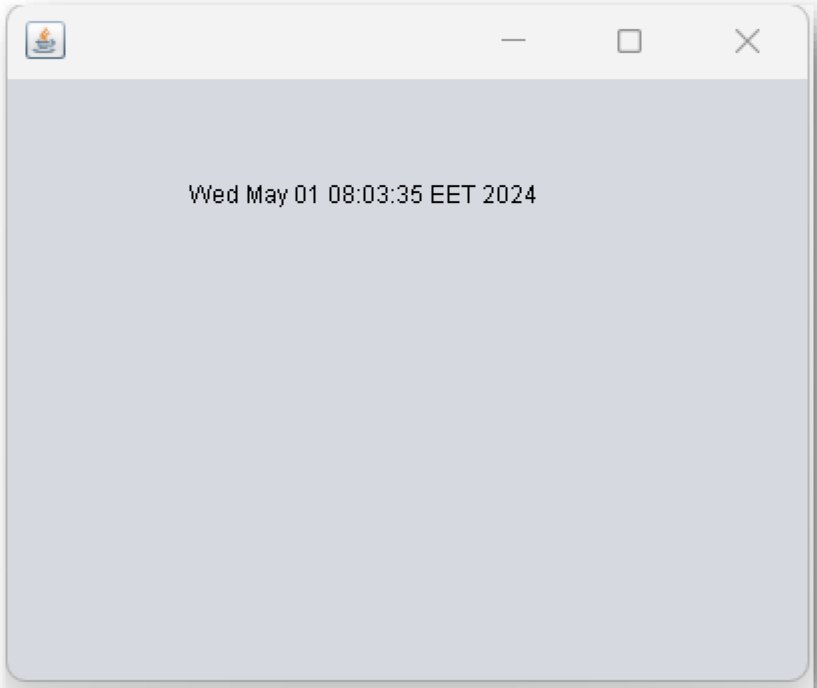


Extending Thread VS. Implementing Runnable

- Choosing between these two is a matter of taste.
- Implementing the Runnable interface:
  - May take more work since we still:
    - Declare a Thread object
    - Call the Thread methods on this object
  - Your class can still extend other class
- Extending the Thread class
  - Easier to implement
  - Your class can no longer extend any other class

Example

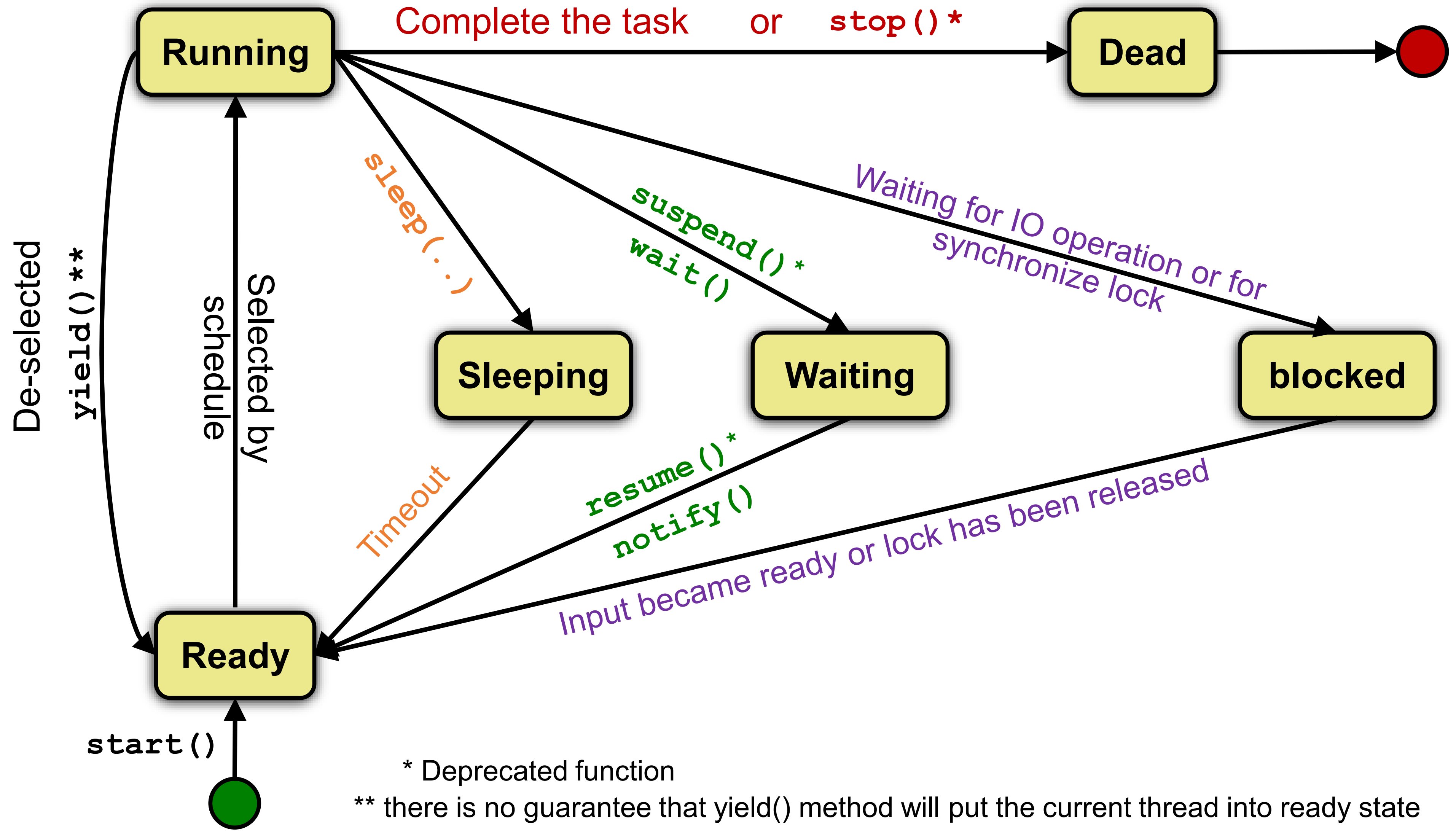
```
public class DateTimeApp extends JFrame implements Runnable{
    Thread th;
    public DateTimeApp() {
        initComponents();
        th = new Thread(this);
        th.start();
    }
    @Override
    public void run(){
        while (true){
            repaint();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) { ex.printStackTrace();}
        }
    }
    @Override
    public void paint(Graphics g) {
        super.paint(g);
        Date d = new Date();
        g.drawString(d.toString(), 100, 100);
    }
```



```
Wed May 01 08:03:35 EET 2024
```

```
public class DateTimeApp extends JFrame implements Runnable{
   Thread th;
   public DateTimeApp (){
     initComponents();
     th = new Thread(this);
     th.start();
   }
   public void paint(Graphics g){
      super.paint(g);
     Date d = new Date();
     g.drawString(d.toString(), 100, 100);
   }
   public void run(){
     while(true){
        try{
          repaint();
          Thread.sleep(1000);   //you'll need to catch an exception here
        }catch(InterruptedException ie){ie.printStackTrace();}
      }
   }
}
```

Thread Life Cycle



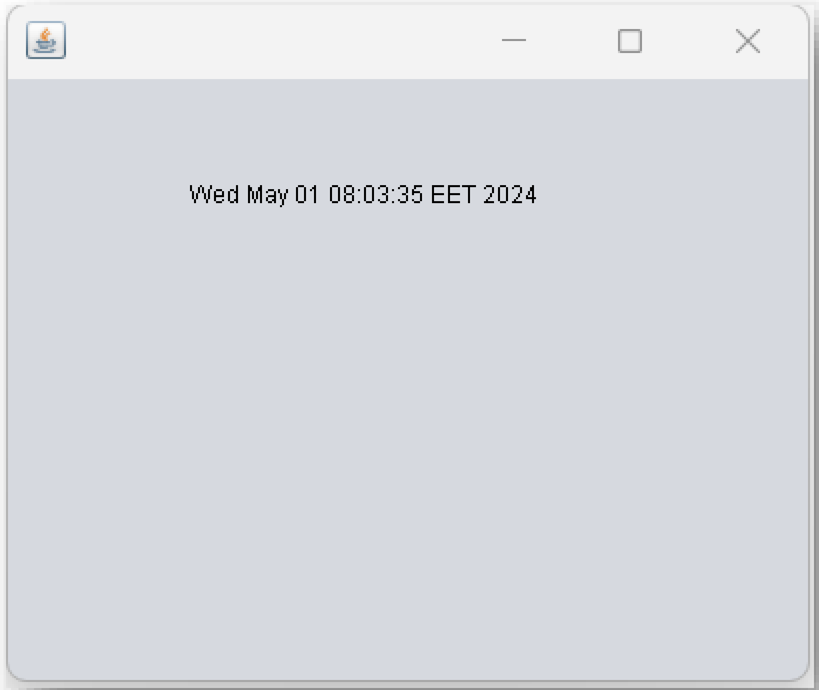
* Deprecated function

** there is no guarantee that yield() method will put the current thread into ready state

Lab Exercises

1. Date and Time JFrame

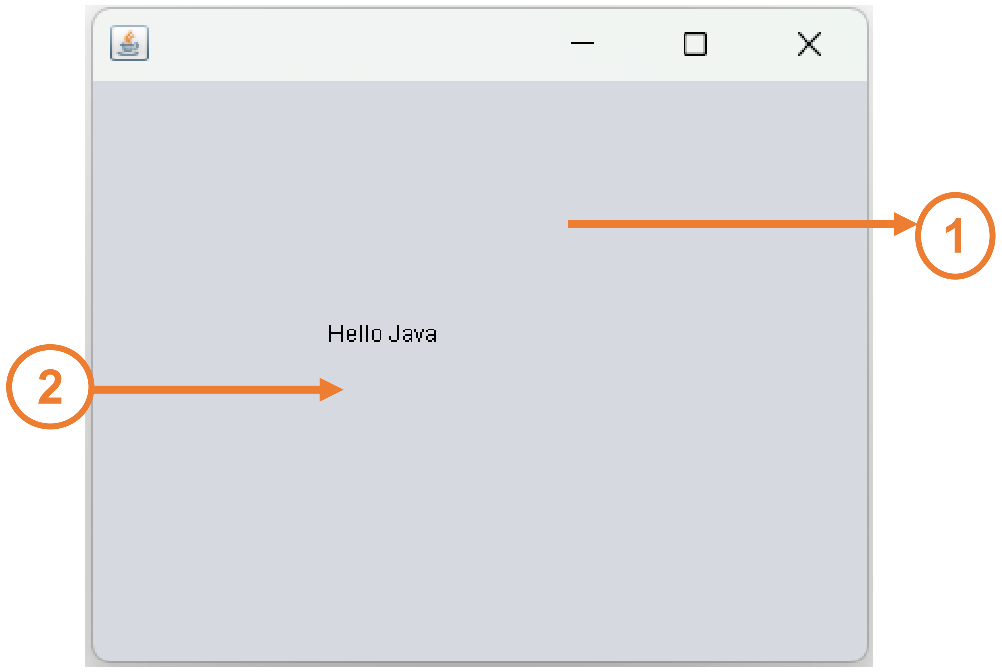
- Create a JFrame that displays date and time on it.



```
Wed May 01 08:03:35 EET 2024
```

2. Text Banner JFrame

- Create a JFrame that displays marquee string on it.


1
Hello Java
2