Android Concurrency : Motivating Java Synchronization & Scheduling Mechanisms

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

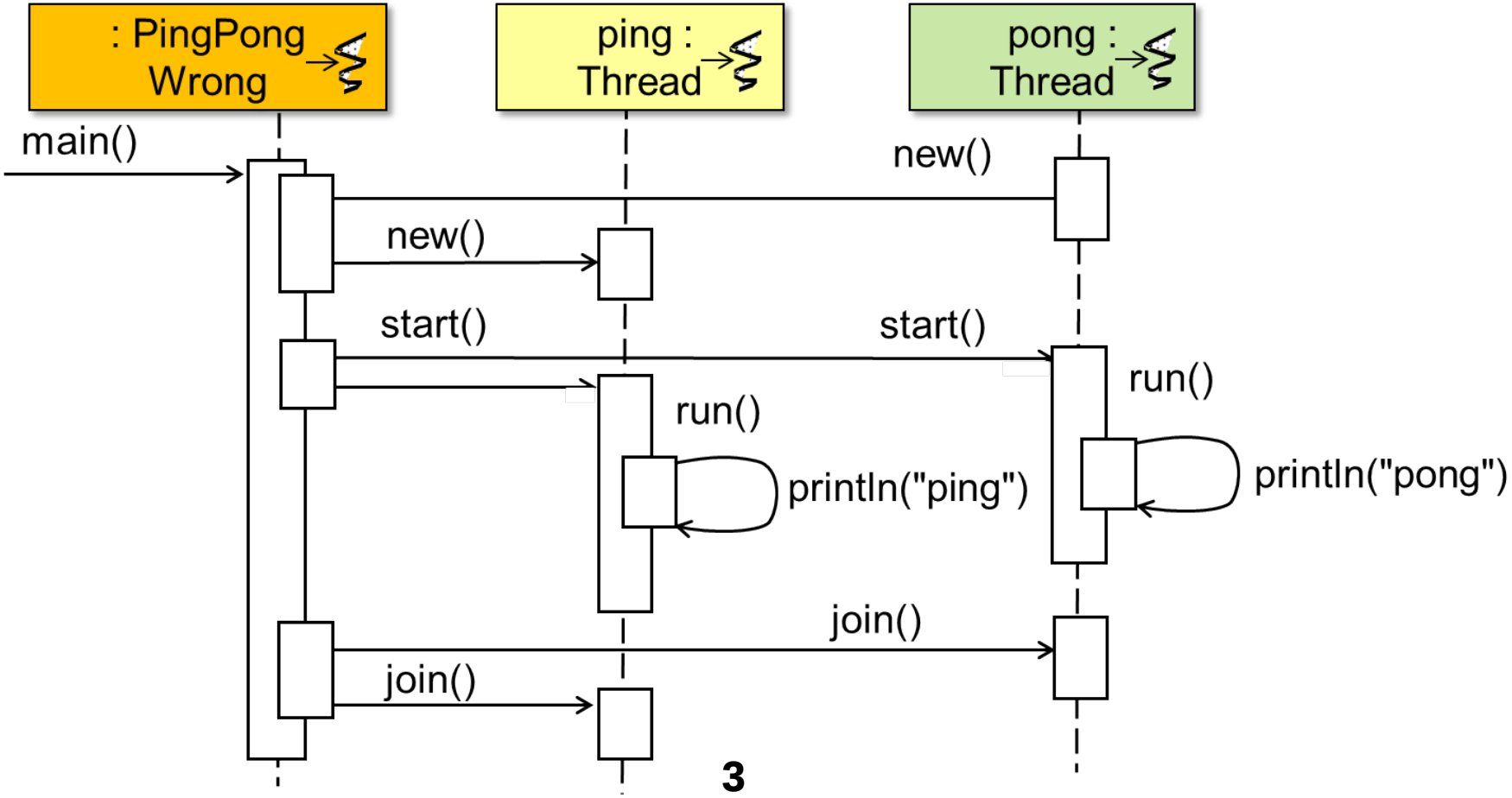
Learning Objectives in this Part of the Module

- Understand the need for synchronization & scheduling mechanisms that
  - Protect against data corruption due to race conditions

en.wikipedia.org/wiki/Race_condition has more on race conditions

Learning Objectives in this Part of the Module

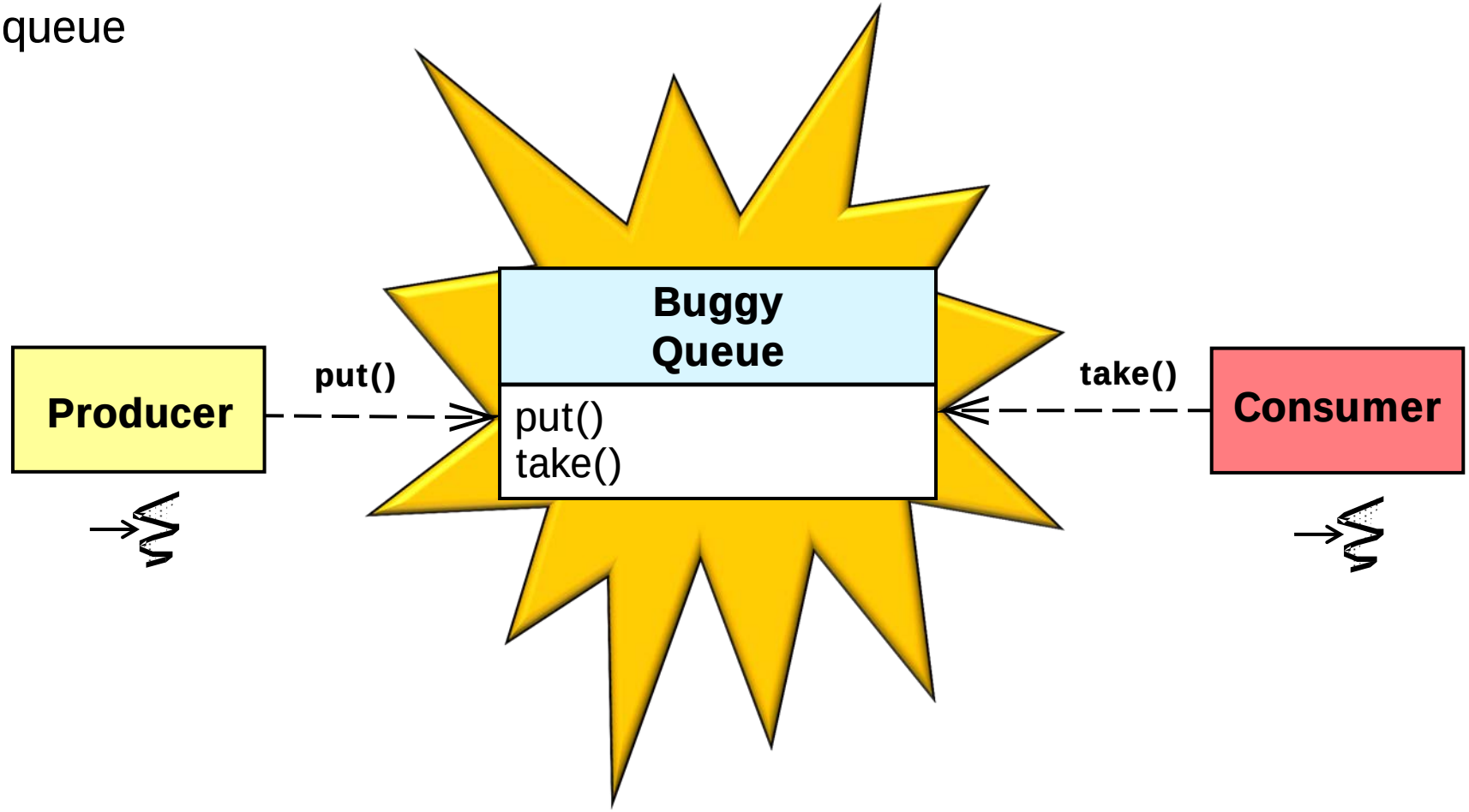
- Understand the need for synchronization & scheduling mechanisms that
  - Protect against data corruption due to race conditions
  - Ensure threads execute in the right order to meet requirements



Motivating Examples

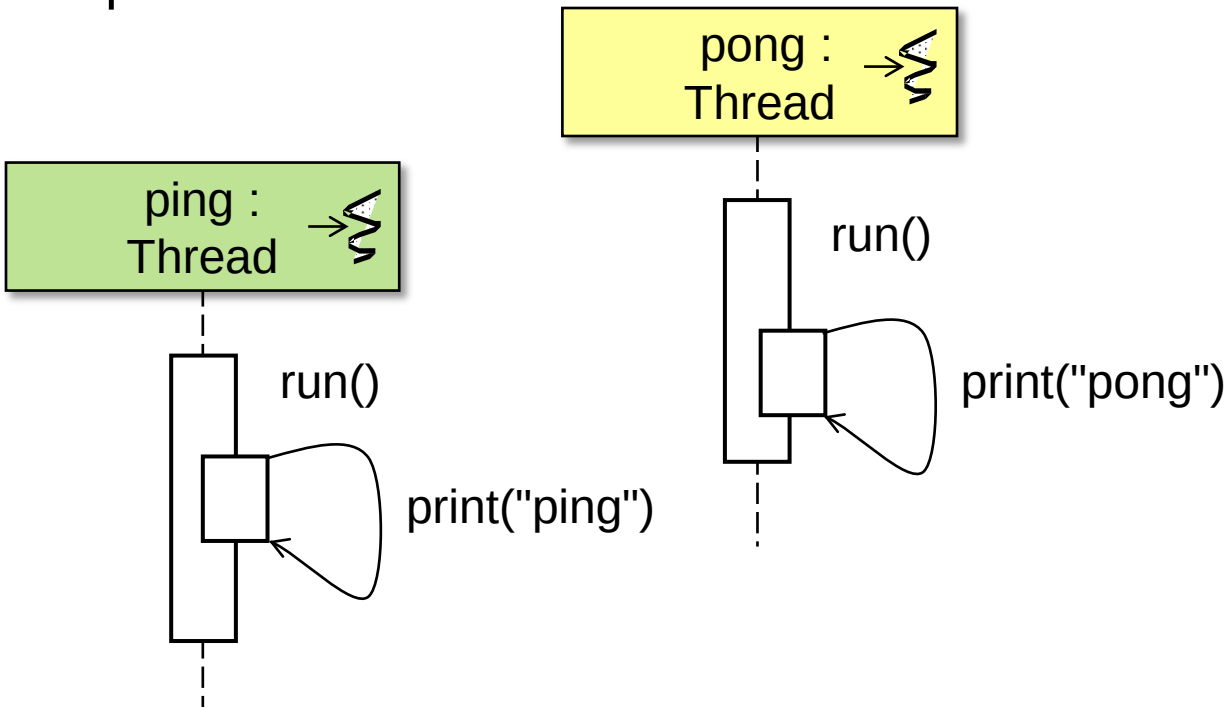
Motivating Examples

1. A program where two Java threads attempt to communicate by passing messages via a shared queue



Motivating Examples

1. A program where two Java threads attempt to communicate by passing messages via a shared queue
2. A program where two Java threads attempt to alternate printing "ping" & "pong" on the computer console




```
% java PingPongWrong
Ready...Set...Go!
Ping!(1)
Ping!(2)
Ping!(3)
Ping!(4)
Ping!(5)
Ping!(6)
Ping!(7)
Ping!(8)
Ping!(9)
Ping!(10)
Pong!(1)
Pong!(2)
Pong!(3)
Pong!(4)
Pong!(5)
Pong!(6)
Pong!(7)
Pong!(8)
Pong!(9)
Pong!(10)
Done!
```

Motivating Examples

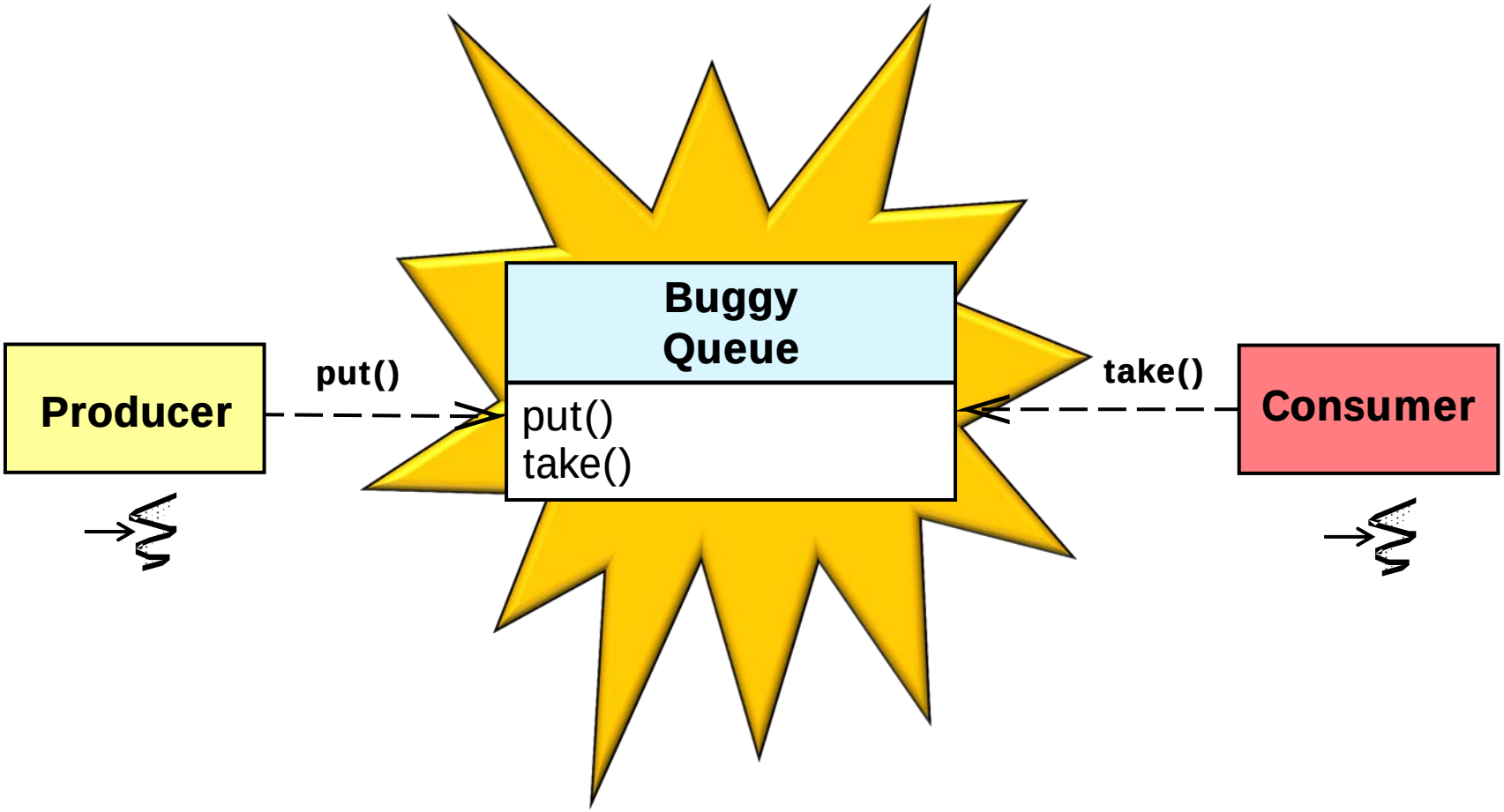
1. A program where two Java threads attempt to communicate by passing messages via a shared queue
2. A program where two Java threads attempt to alternate printing "ping" & "pong" on the computer console

We'll cover solutions to these examples throughout the rest of the module

Example 1: A Buggy Producer/ Consumer Program

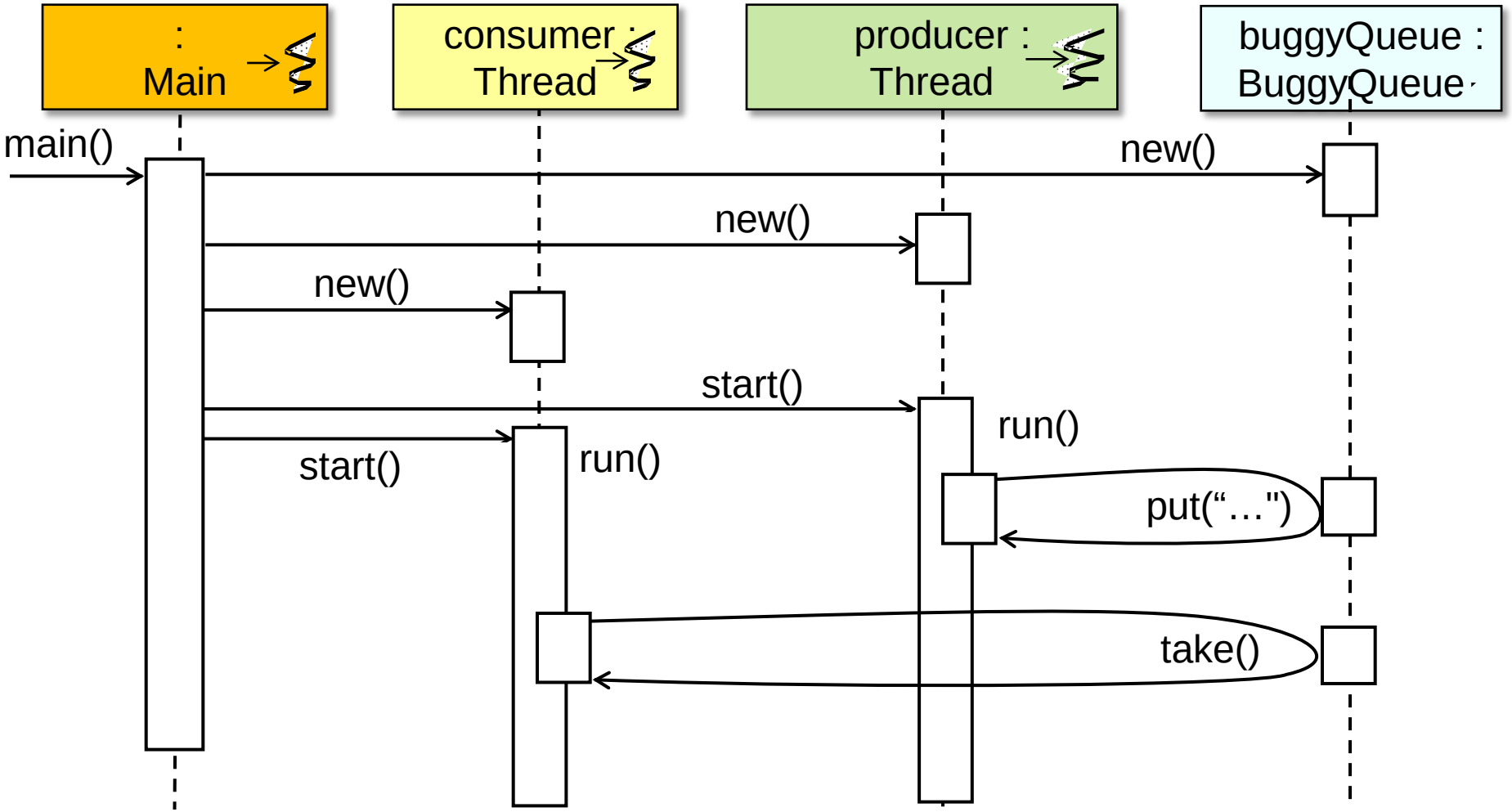
Buggy Producer/Consumer Program Overview

- A simple concurrent producer/consumer program that attempts to pass messages via an object modeled on the Java ArrayBlockingQueue class



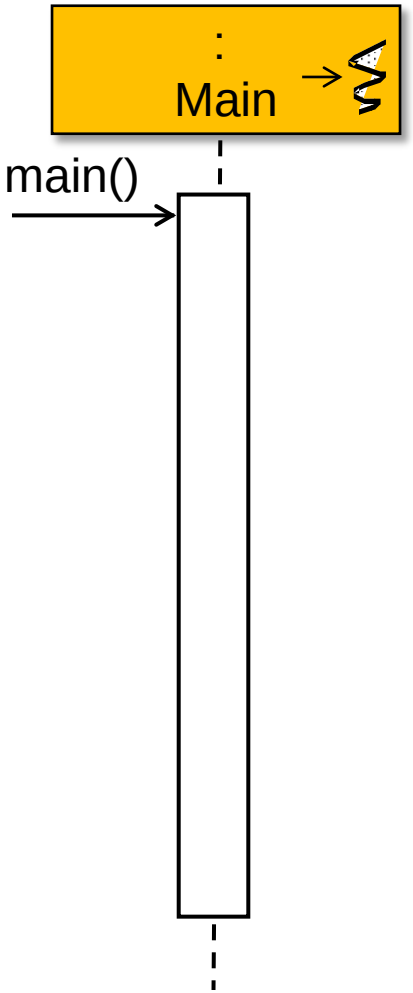
Design of the Buggy Producer/Consumer Program

- UML sequence diagram showing the design of the buggy producer/consumer



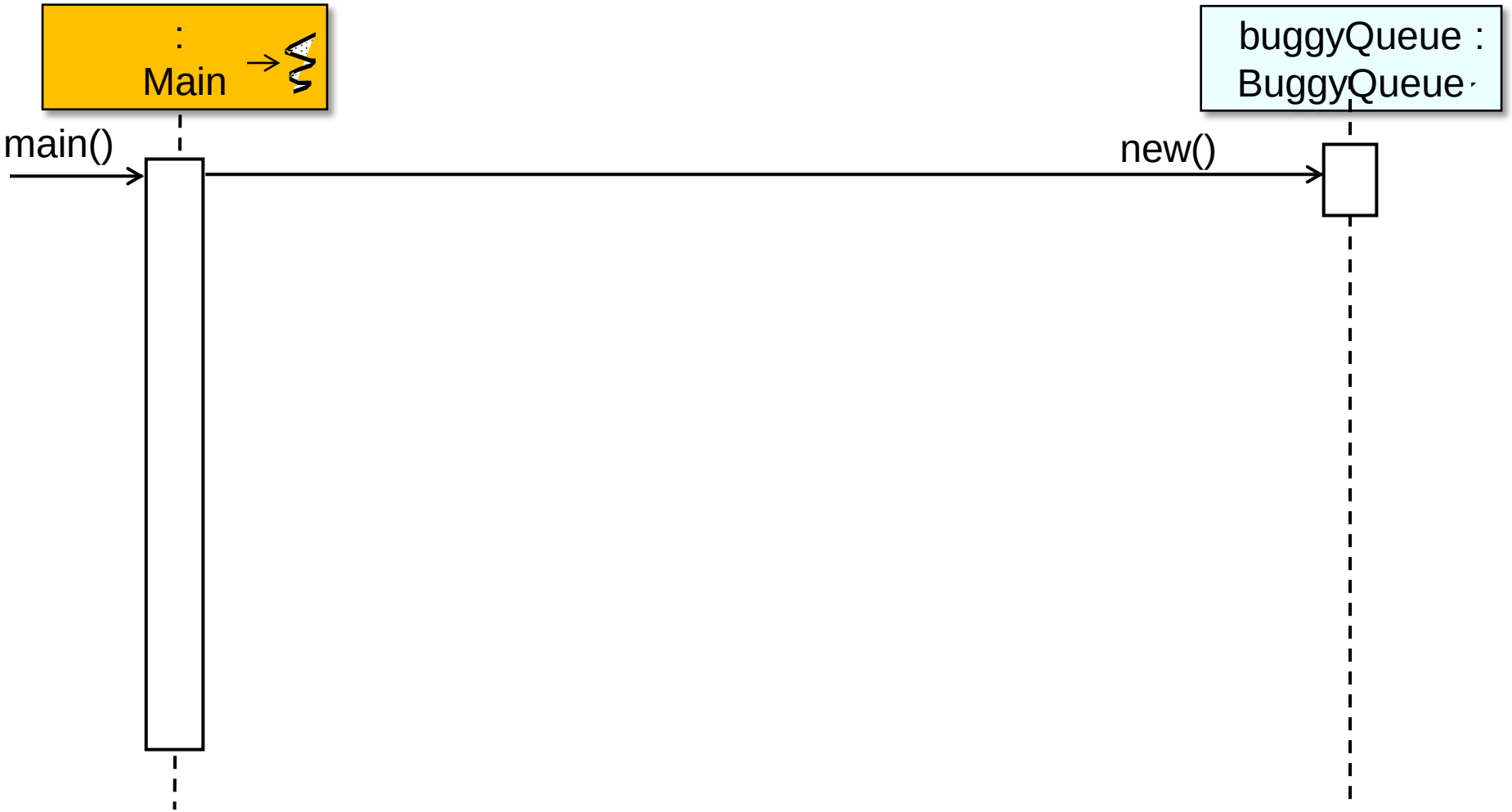
Design of the Buggy Producer/Consumer Program

- UML sequence diagram showing the design of the buggy producer/consumer



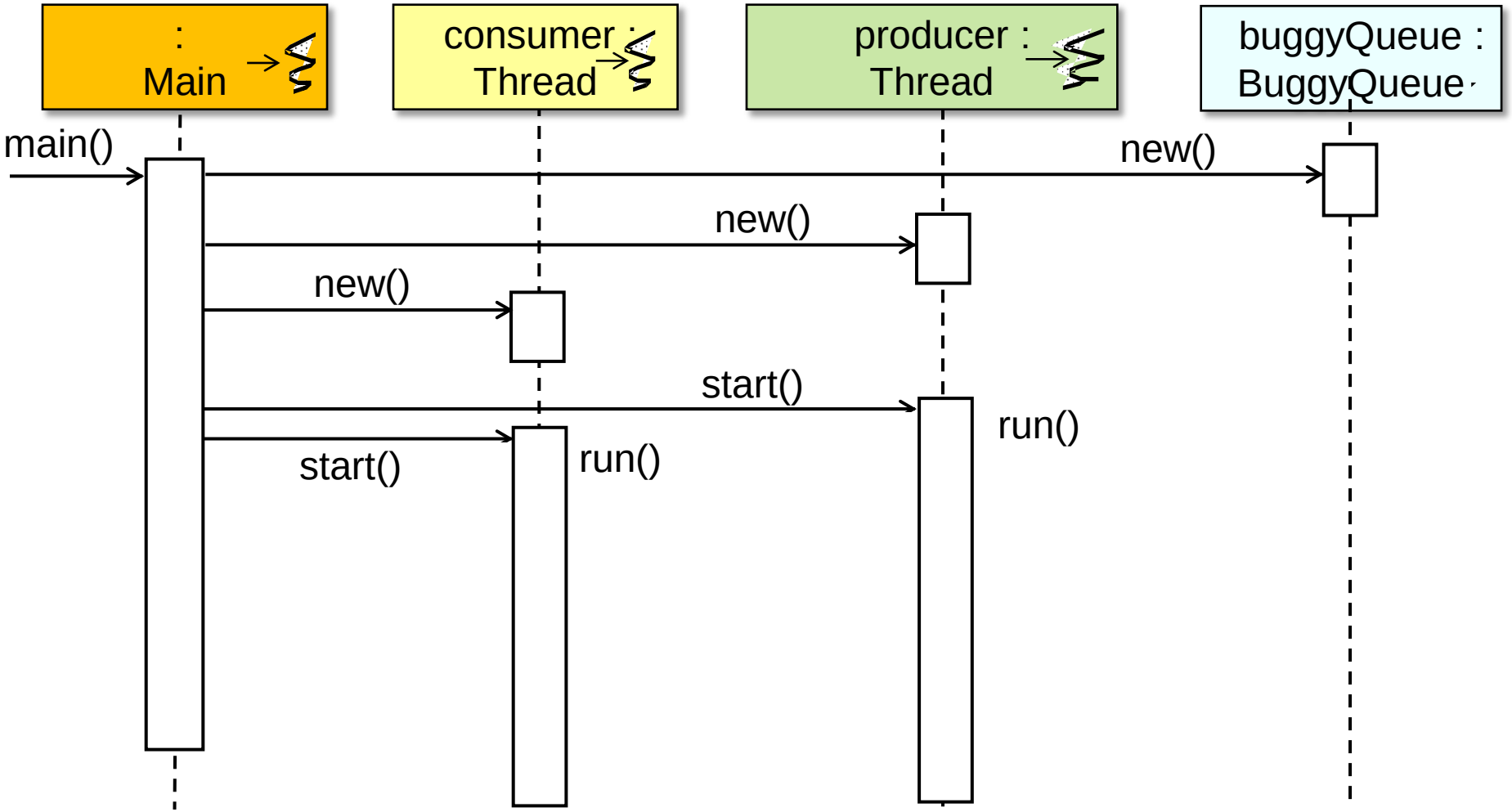
Design of the Buggy Producer/Consumer Program

- UML sequence diagram showing the design of the buggy producer/consumer



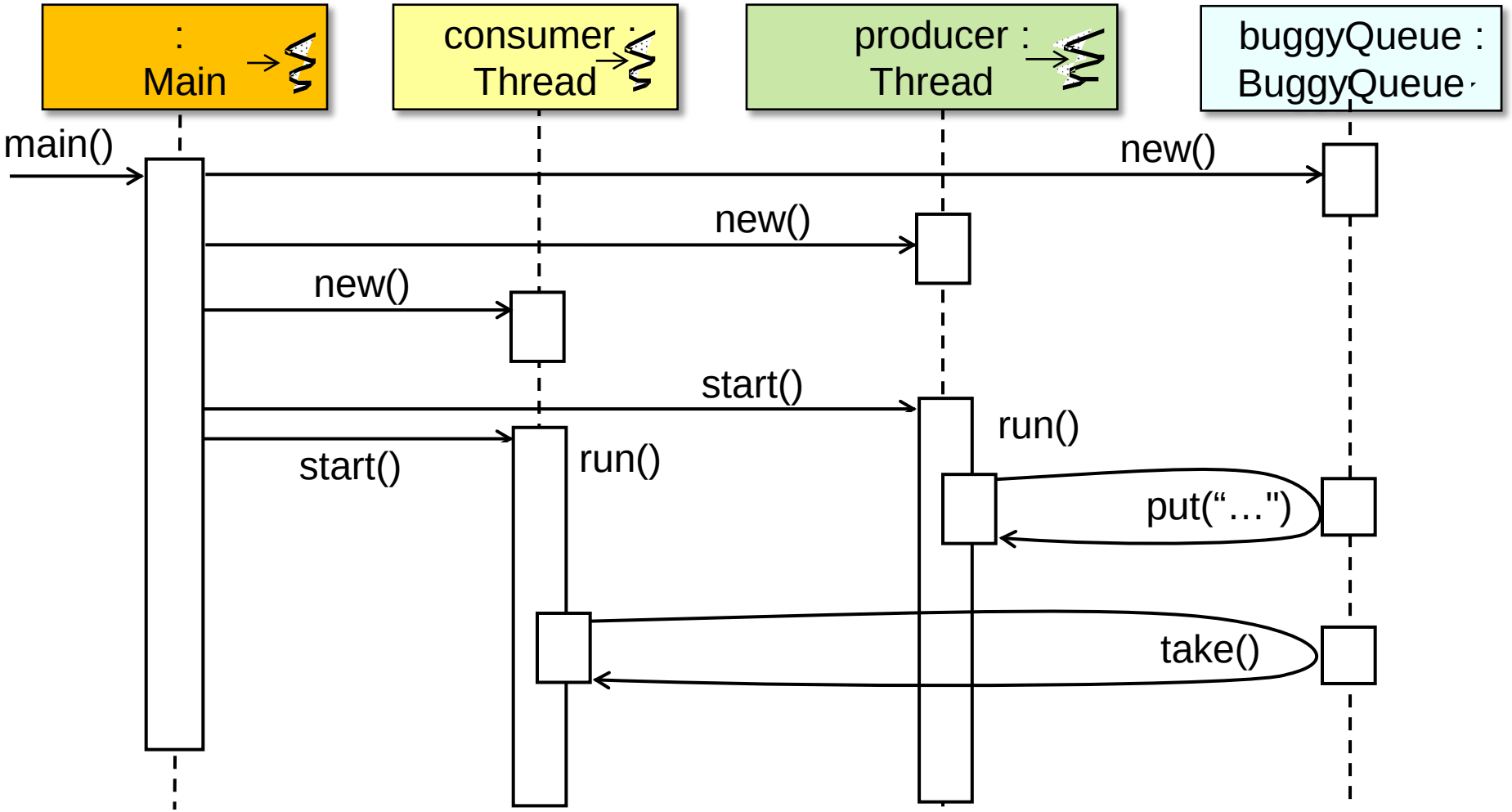
Design of the Buggy Producer/Consumer Program

- UML sequence diagram showing the design of the buggy producer/consumer



Design of the Buggy Producer/Consumer Program

- UML sequence diagram showing the design of the buggy producer/consumer



Implementation of the Buggy Producer/Consumer

- Implementation of the BuggyQueue class

```
static class BuggyQueue {
  private List<String> mQ = new ArrayList<String>();


  public void put(String msg){ mQ.add(msg); }



  public String take(){ return mQ.remove(0); }
}
```

See github.com/douglascraigschmidt/POSA-14/tree/master/ex/BuggyQueue

Implementation of the Buggy Producer/Consumer

- Implementation of the BuggyQueue class

```
static class BuggyQueue {
  private List<String> mQ = new ArrayList<String>();

  public void put(String msg){ mQ.add(msg); }



  public String take(){ return mQ.remove(0); }
}
```

Resizable-array implementation

docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html has more

Implementation of the Buggy Producer/Consumer

- Implementation of the BuggyQueue class

```
static class BuggyQueue {
  private List<String> mQ = new ArrayList<String>();


  public void put(String msg){ mQ.add(msg); }



  public String take(){ return mQ.remove(0); }
}
```

Implementation of the Buggy Producer/Consumer

- Implementation of the BuggyQueue class

```
static class BuggyQueue {
  private List<String> mQ = new ArrayList<String>();


  public void put(String msg){ mQ.add(msg); }


  public String take(){ return mQ.remove(0); }
}
```

Add & remove strings into/from the queue

Implementation of the Buggy Producer/Consumer

- Implementation of the Main class

```
public static void main(String argv[]) {
  final BuggyQueue buggyQueue = new BuggyQueue();

  Thread producer =
     new Thread(new Runnable(){
                public void run(){
                  for(int i = 0; i < mMaxIterations; i++)
                    buggyQueue.put(Integer.toString(i));
                }});
     Thread consumer =
       new Thread(new Runnable(){
                  public void run(){
                    for(int i = 0; i < mMaxIterations; i++)
                      System.out.println(buggyQueue.take());
                }});

  producer.start();
  consumer.start();
  ...
```

Implementation of the Buggy Producer/Consumer

- Implementation of the Main class

Create producer & consumer threads

```
public static void main(String argv[]) {
  final BuggyQueue buggyQueue = new BuggyQueue();

  Thread producer =
     new Thread(new Runnable(){
                public void run(){
                  for(int i = 0; i < mMaxIterations; i++)
                    buggyQueue.put(Integer.toString(i));
                }});
     Thread consumer =
       new Thread(new Runnable(){
                  public void run(){
                    for(int i = 0; i < mMaxIterations; i++)
                      System.out.println(buggyQueue.take());
                }});

  producer.start();
  consumer.start();
  ...
```

Implementation of the Buggy Producer/Consumer

- Implementation of the Main class

```
public static void main(String argv[]) {
  final BuggyQueue buggyQueue = new BuggyQueue();

  Thread producer =
     new Thread(new Runnable(){
                public void run(){
                  for(int i = 0; i < mMaxIterations; i++)
                    buggyQueue.put(Integer.toString(i));
                }});
     Thread consumer =
       new Thread(new Runnable(){
                  public void run(){
                    for(int i = 0; i < mMaxIterations; i++)
                      System.out.println(buggyQueue.take());
                }});

  producer.start();
  consumer.start();
  ...
```

Start producer & consumer threads

Implementation of the Buggy Producer/Consumer

- Implementation of the Main class

```
public static void main(String argv[]) {
  final BuggyQueue buggyQueue = new BuggyQueue();

  Thread producer =
     new Thread(new Runnable(){
                public void run(){
                  for(int i = 0; i < mMaxIterations; i++)
                    buggyQueue.put(Integer.toString(i));
                }});
     Thread consumer =
       new Thread(new Runnable(){
                  public void run(){
                    for(int i = 0; i < mMaxIterations; i++)
                      System.out.println(buggyQueue.take());
                }});

  producer.start();
  consumer.start();
  ...
```

Produce & consume messages concurrently

Implementation of the Buggy Producer/Consumer

- Implementation of the Main class

```
public static void main(String argv[]) {
  final BuggyQueue buggyQueue = new BuggyQueue();

  Thread producer =
     new Thread(new Runnable(){
                public void run(){
                  for(int i = 0; i < mMaxIterations; i++)
                    buggyQueue.put(Integer.toString(i));
                }});
     Thread consumer =
       new Thread(new Runnable(){
                  public void run(){
                    for(int i = 0; i < mMaxIterations; i++)
                      System.out.println(buggyQueue.take());
                }});

  producer.start();
  consumer.start();
  ...
```

Produce & consume messages concurrently

Implementation of the Buggy Producer/Consumer

- Implementation of the Main class

```
public static void main(String argv[]) {
  final BuggyQueue buggyQueue = new BuggyQueue();

  Thread producer =
     new Thread(new Runnable(){
                public void run(){
                  for(int i = 0; i < mMaxIterations; i++)
                    buggyQueue.put(Integer.toString(i));
                }});
     Thread consumer =
       new Thread(new Runnable(){
                  public void run(){
                    for(int i = 0; i < mMaxIterations; i++)
                      System.out.println(buggyQueue.take());
                }});

  producer.start();
  consumer.start();
  ...
```

Key question: What output will this code produce?

Implementation of the Buggy Producer/Consumer

- Implementation of the Main class

```
public static void
  final BuggyQueue

  Thread producer =
     new Thread(new
                pub
                  f

                }})
     Thread consume
       new Thread(n
```

```
Exception in thread "Thread-1"
java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
        at java.util.ArrayList.RangeCheck(ArrayList.java:547)
        at java.util.ArrayList.remove(ArrayList.java:387)
        at Main$BuggyQueue.take(Main.java:16)
        at Main$2.run(Main.java:34)
        at java.lang.Thread.run(Thread.java:662)
```

Implementation of the Buggy Producer/Consumer

- Implementation of the BuggyQueue class

```
static class BuggyQueue {
  private List<String> mQ = new ArrayList<String>();


  public void put(String msg){ mQ.add(msg); }
```

There's no protection against critical sections being run by multiple threads concurrently

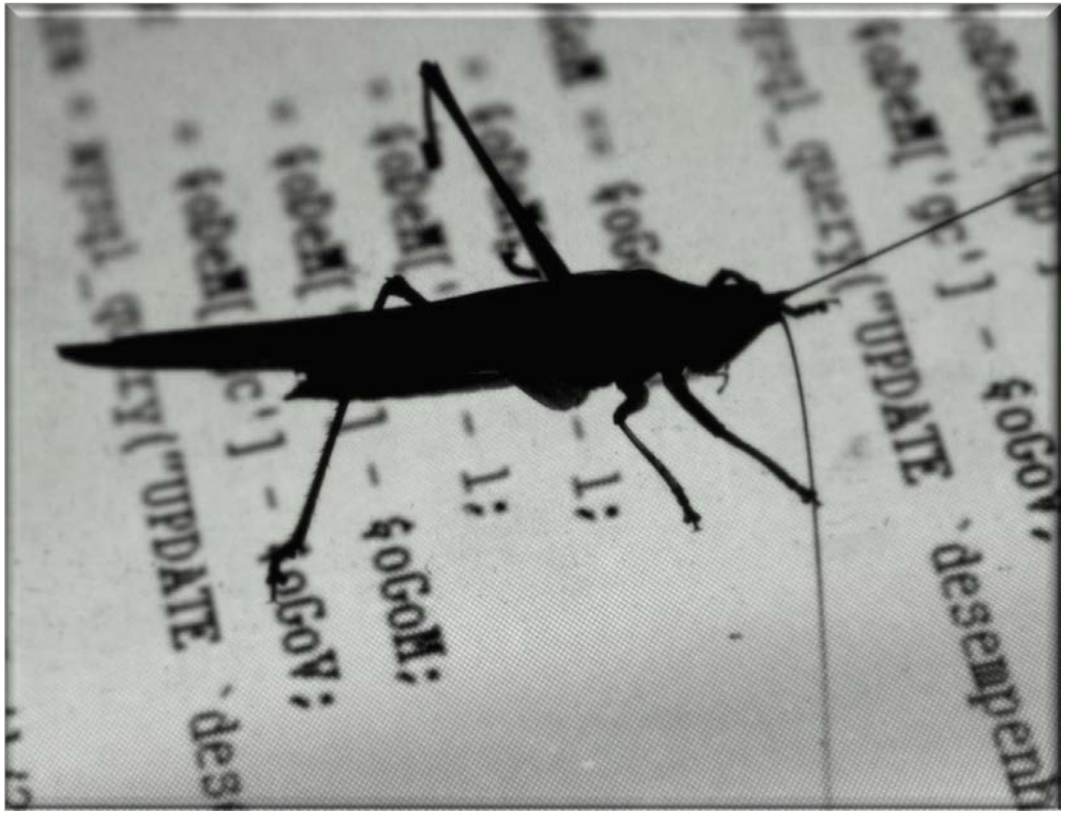
```
  public String take(){ return mQ.remove(0); }
}
```

Note that this implementation is not synchronized. If multiple threads access an `ArrayList` instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.)

docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html has more

Evaluating the Buggy Producer/Consumer

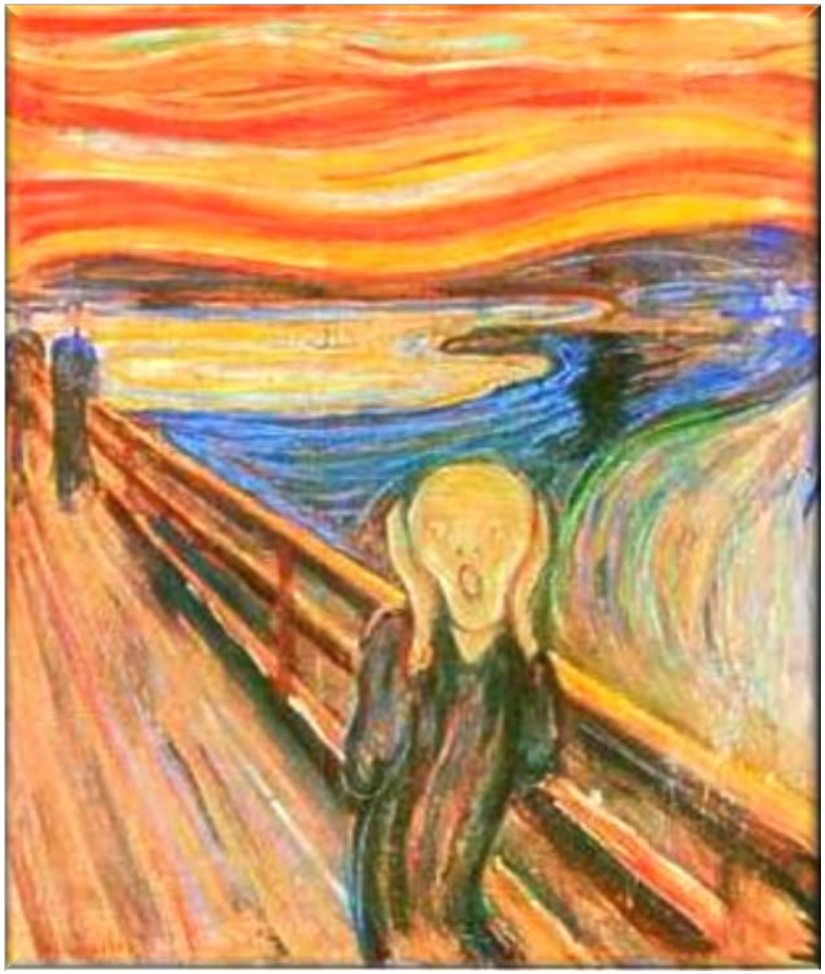
- The race conditions shown in the BuggyQueue are hard to detect & debug, due to the inherent complexities of concurrency



See earlier part on “Challenges of Concurrency”

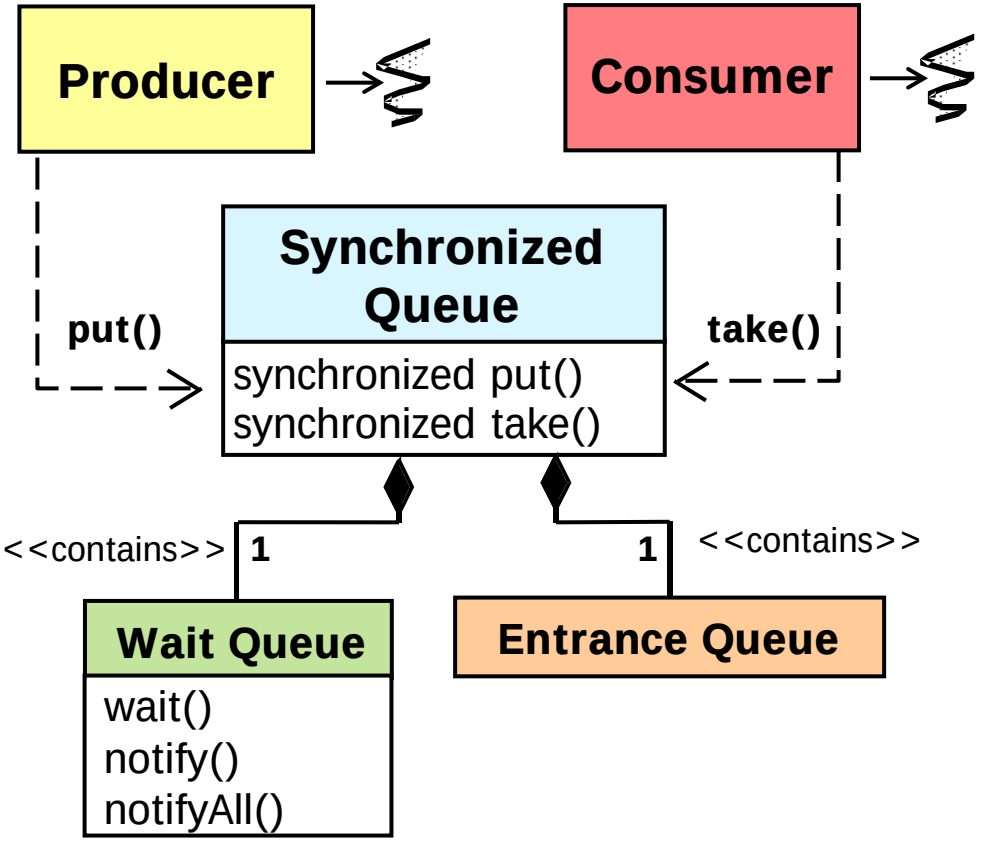
Evaluating the Buggy Producer/Consumer

- The race conditions shown in the BuggyQueue are hard to detect & debug, due to the inherent complexities of concurrency
- Development & quality assurance therefore become tedious, error-prone, & non-portable



Evaluating the Buggy Producer/Consumer

- The race conditions shown in the BuggyQueue are hard to detect & debug, due to the inherent complexities of concurrency
- Development & quality assurance therefore become tedious, error-prone, & non-portable
- We'll fix these problems by applying various Java synchronization & scheduling mechanisms

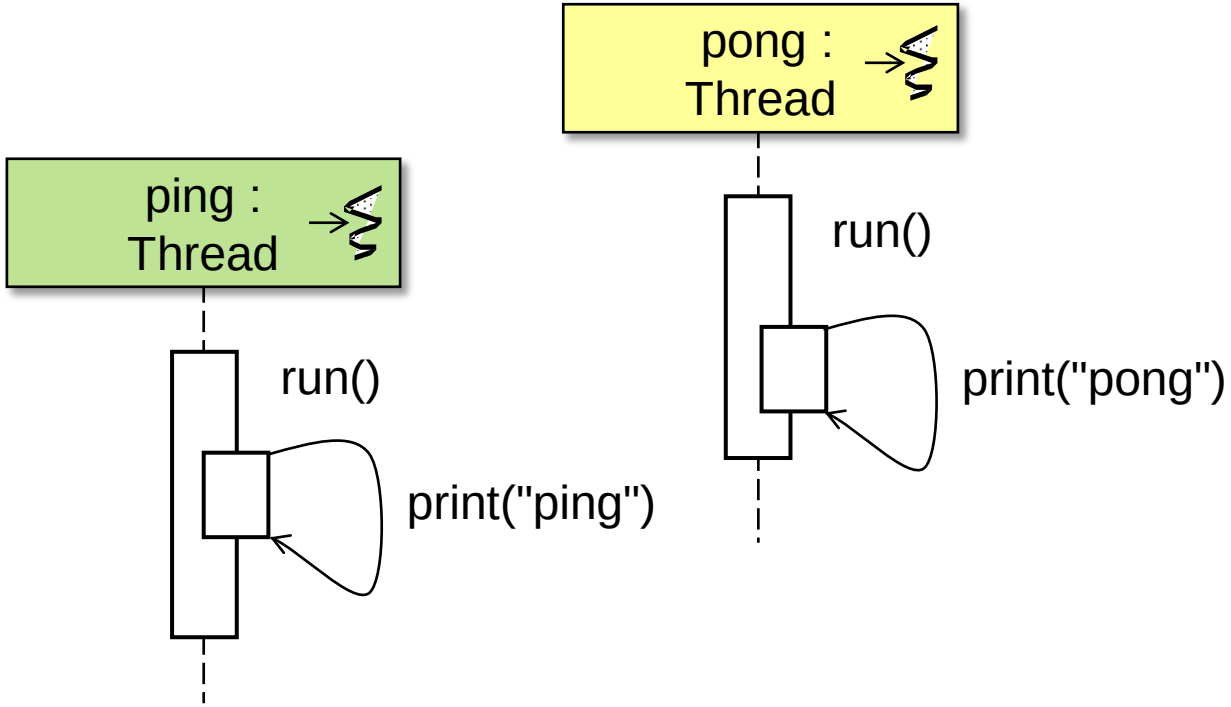


See upcoming parts on "Java ConditionObject" & "Java Built-in Monitor Objects"

Example 2: A Buggy Ping-Pong Program

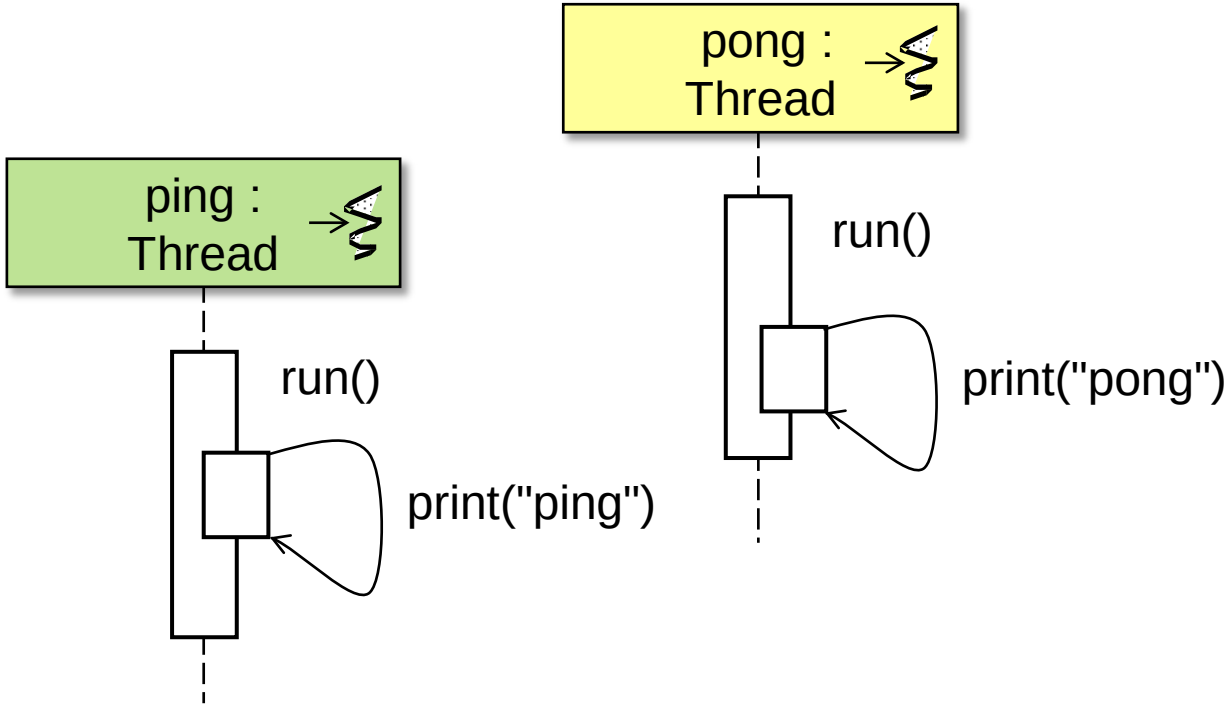
Buggy Ping-Pong Program Overview

- This program has two Java threads that attempt to alternate printing “Ping” & “Pong” on the console



Buggy Ping-Pong Program Overview

- This program has two Java threads that attempt to alternate printing “Ping” & “Pong” on the console

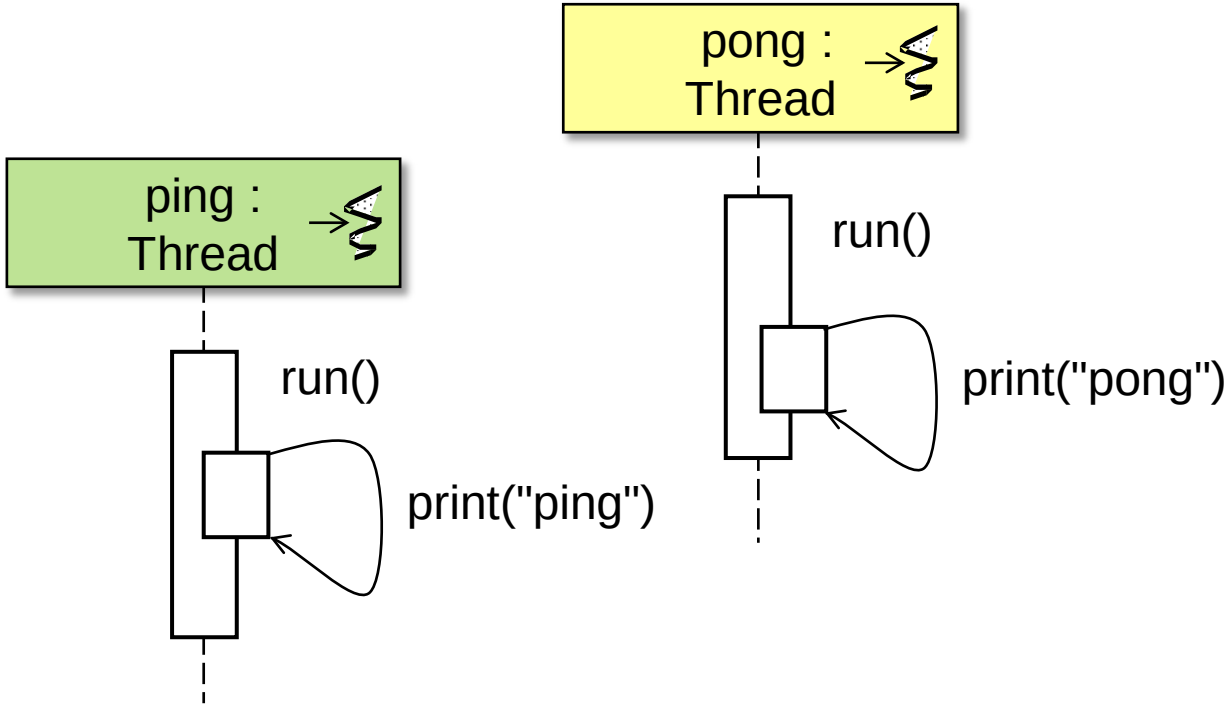



```
% java PingPongWrong
Ready...Set...Go!
Ping!(1)
Ping!(2)
Ping!(3)
Ping!(4)
Ping!(5)
Ping!(6)
Ping!(7)
Ping!(8)
Ping!(9)
Ping!(10)
Pong!(1)
Pong!(2)
Pong!(3)
Pong!(4)
Pong!(5)
Pong!(6)
Pong!(7)
Pong!(8)
Pong!(9)
Pong!(10)
Done!
```

See developer.android.com/training/multiple-threads/communicate-ui.html

Buggy Ping-Pong Program Overview

- This program has two Java threads that attempt to alternate printing “Ping” & “Pong” on the console

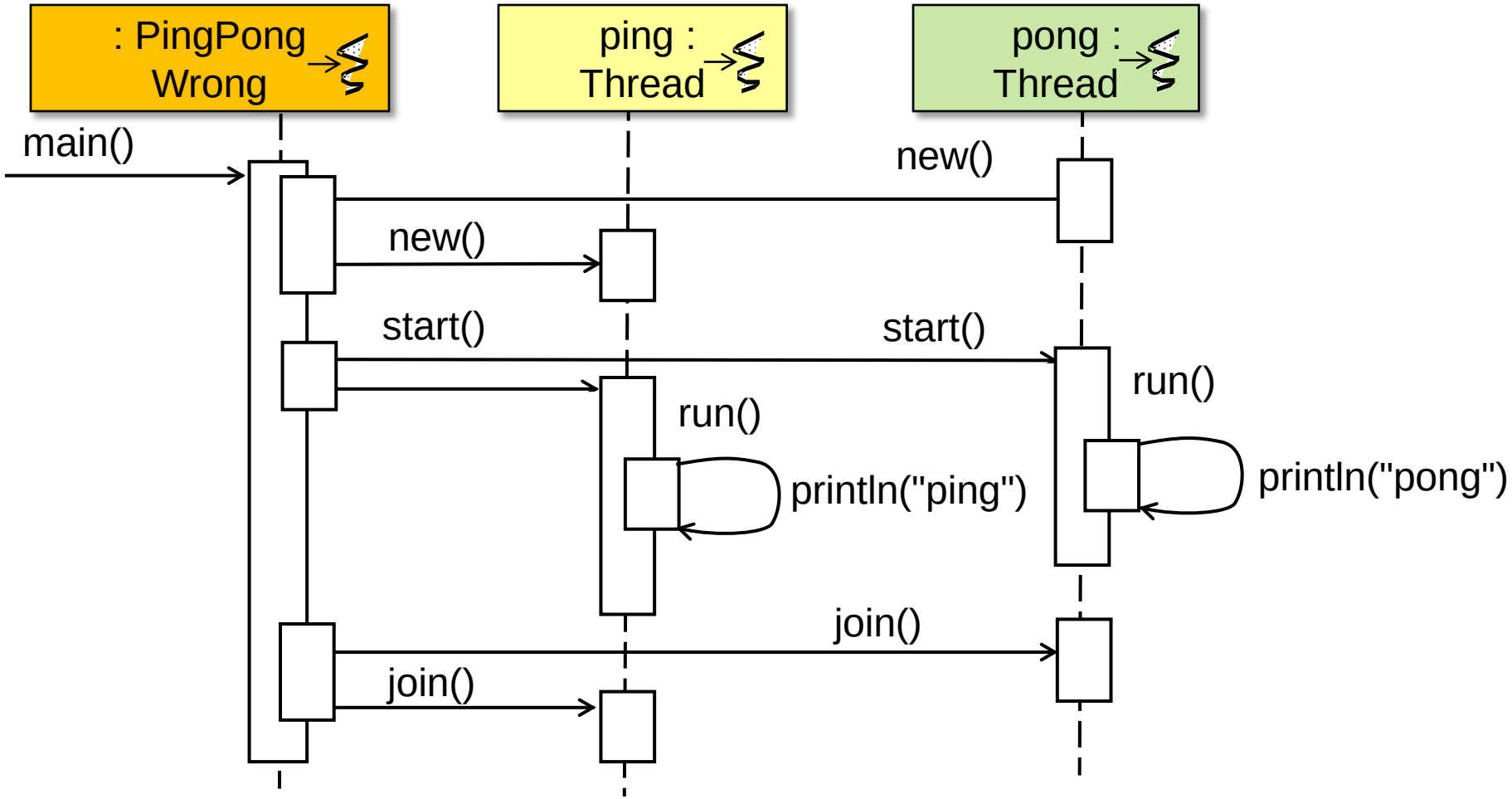



```
% java PingPongWrong
Ready...Set...Go!
Ping!(1)
Ping!(2)
Ping!(3)
Ping!(4)
Ping!(5)
Ping!(6)
Ping!(7)
Ping!(8)
Ping!(9)
Ping!(10)
Pong!(1)
Pong!(2)
Pong!(3)
Pong!(4)
Pong!(5)
Pong!(6)
Pong!(7)
Pong!(8)
Pong!(9)
Pong!(10)
Done!
```

See upcoming module on “Android Concurrency Frameworks”

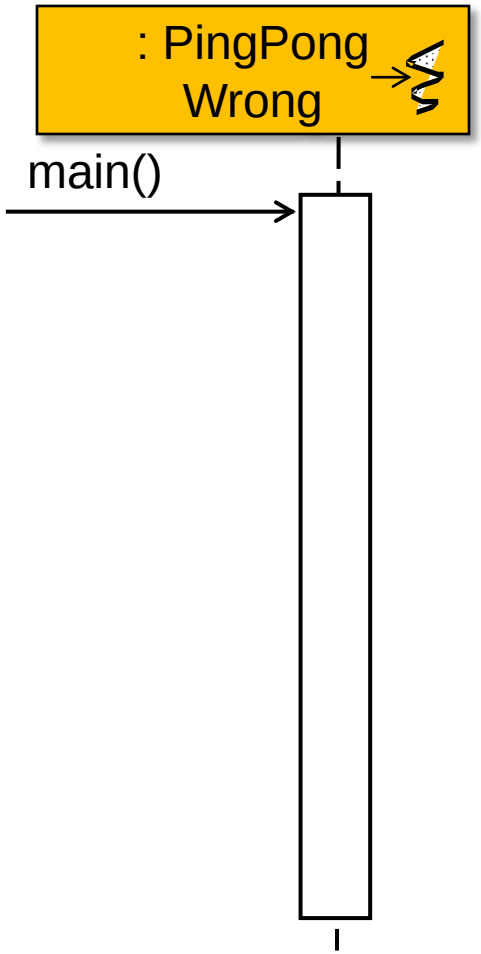
Design of the Buggy Ping-Pong Program

- UML sequence diagram showing the design of the buggy ping-pong program



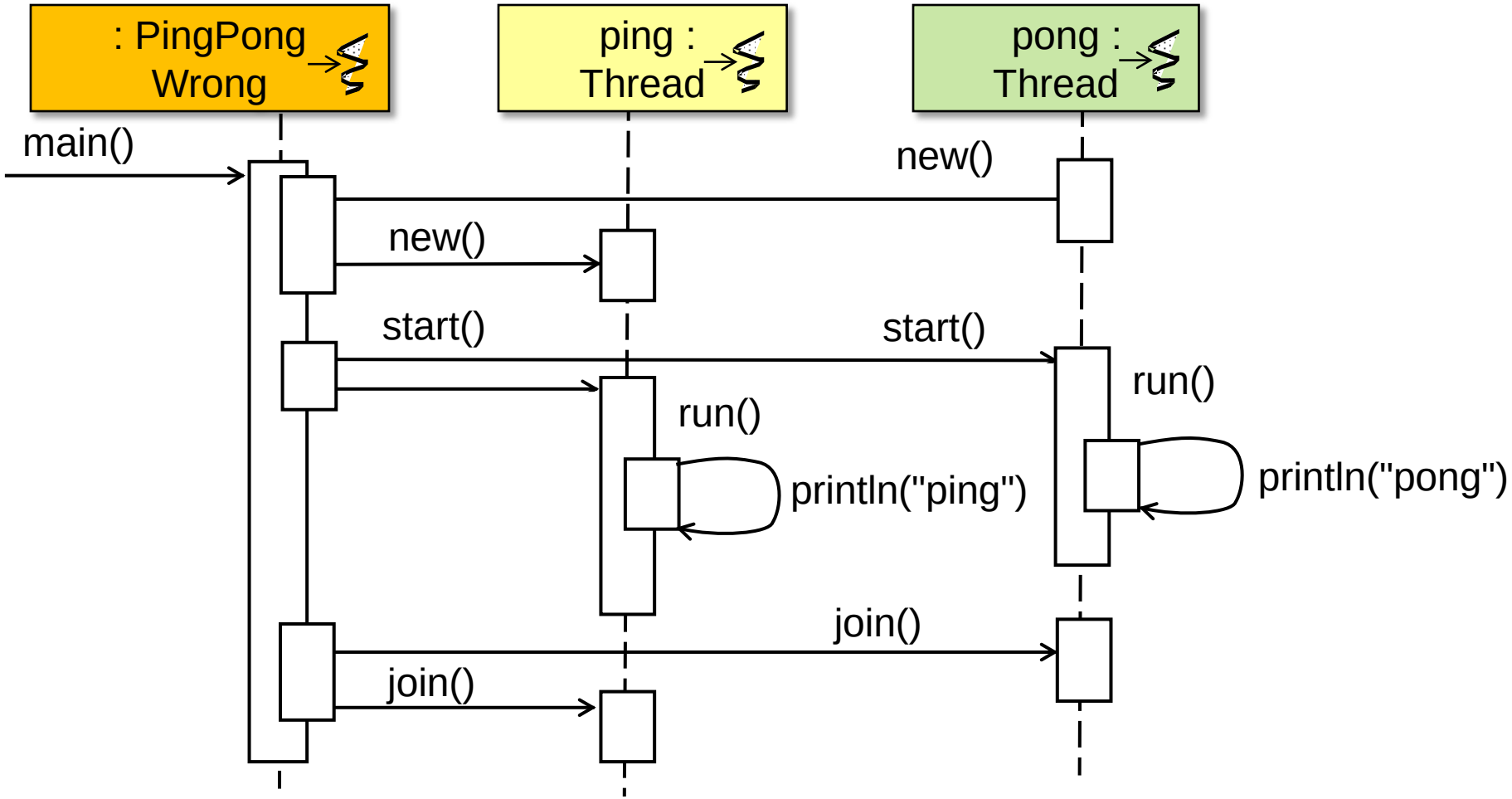
Design of the Buggy Ping-Pong Program

- UML sequence diagram showing the design of the buggy ping-pong program



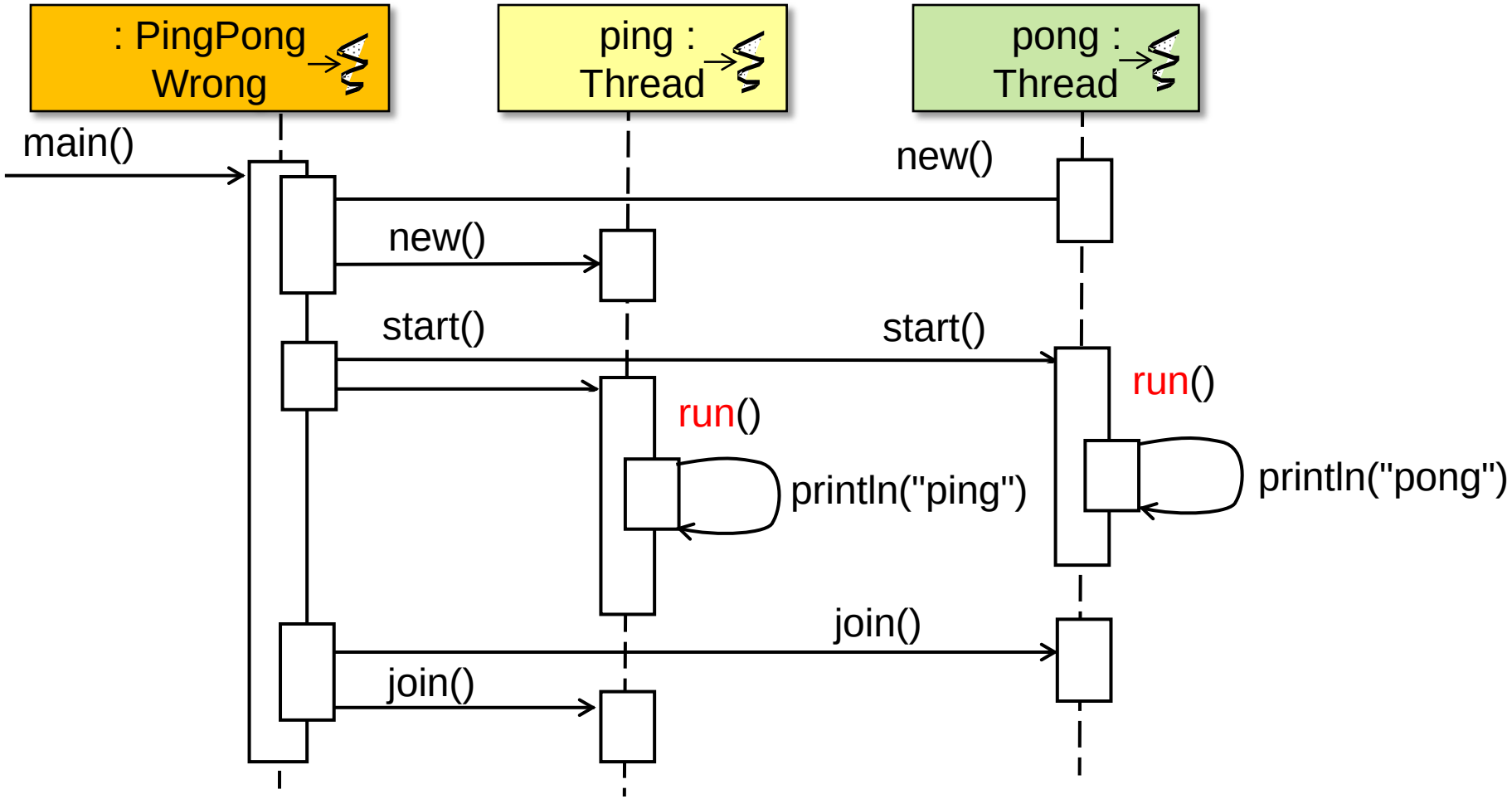
Design of the Buggy Ping-Pong Program

- UML sequence diagram showing the design of the buggy ping-pong program



Design of the Buggy Ping-Pong Program

- UML sequence diagram showing the design of the buggy ping-pong program



See github.com/douglascraigschmidt/POSA-14/tree/master/ex/PingPong/wrong

Implementation of the Buggy Ping-Pong Program

- Implementation of the PingPongWrong class

```
public class PingPongWrong {
  public static int mMaxIterations = 10;
```

Number of iterations to play ping-pong

Implementation of the Buggy Ping-Pong Program

- Implementation of the PlayPingPongThread class

```
public class PingPongWrong {
  ...
  public static class PlayPingPongThread extends Thread {

    private String mStringToPrint;

    public PlayPingPongThread (String stringToPrint) {
      this.mStringToPrint = stringToPrint;
    }
  }
  ...
```

Extends Thread & is used to run the ping-pong logic concurrently

Implementation of the Buggy Ping-Pong Program

- Implementation of the PlayPingPongThread class

```
public class PingPongWrong {
  ...
  public static class PlayPingPongThread extends Thread {



    private String mStringToPrint;
```

Define/set the string to print for each ping & pong operation

```
    public PlayPingPongThread (String stringToPrint) {
      this.mStringToPrint = stringToPrint;
    }
  }
  ...
```

Implementation of the Buggy Ping-Pong Program

- Implementation of the PlayPingPongThread class

```
public class PingPongWrong {
  ...
  public static class PlayPingPongThread extends Thread {
    ...
    public void run() {


      for (int loopsDone = 1;
           loopsDone <= mMaxIterations;
           ++loopsDone)
        System.out.println(mStringToPrint
                           + "(" + loopsDone + ")");
    }

  }
 ...
}
```

Hook method called in each thread

Implementation of the Buggy Ping-Pong Program

- Implementation of the PlayPingPongThread class

```
public class PingPongWrong {
  ...
  public static class PlayPingPongThread extends Thread {
    ...
    public void run() {


      for (int loopsDone = 1;
           loopsDone <= mMaxIterations;
           ++loopsDone)
        System.out.println(mStringToPrint
                           + "(" + loopsDone + ")");
    }

  }
 ...
}
```

Implementation of the Buggy Ping-Pong Program

- Implementation of the PlayPingPongThread class

```
public class PingPongWrong {
  ...
  public static class PlayPingPongThread extends Thread {
    ...
    public void run() {


      for (int loopsDone = 1;
           loopsDone <= mMaxIterations;
           ++loopsDone)
        System.out.println(mStringToPrint
                           + "(" + loopsDone + ")");
    }

  }
 ...
}
```

Print out the iteration

Note lack of proper synchronization!

See upcoming part on “Java Synchronization & Scheduling Example”

Implementation of the Buggy Ping-Pong Program

- Implementation of the PlayPingPongThread class

```
public class PingPongWrong {
  ...
  public static class PlayPingPongThread extends Thread {
    ...
    public void run() {


      for (int loopsDone = 1;
           loopsDone <= mMaxIterations;
           ++loopsDone)
        System.out.println(mStringToPrint
                           + "(" + loopsDone + ")");
    }
```

Exit method & thread when loop is done

```
 ...
}
```

Implementation of the Buggy Ping-Pong Program

- Implementation of the PingPongWrong class

```
public class PingPongWrong {
  ...
  public static void main(String[] args) {
    ...
    System.out.println("Ready...Set...Go!");

    PlayPingPongThread ping =
      new PlayPingPongThread("Ping!");
    PlayPingPongThread pong =
      new PlayPingPongThread("Pong!");

    ping.start();
    pong.start();

    ping.join();
    pong.join();

    System.out.println("Done!");
  ...
```

Main entry point into this app

Implementation of the Buggy Ping-Pong Program

- Implementation of the PingPongWrong class

```
public class PingPongWrong {
  ...
  public static void main(String[] args) {
    ...
    System.out.println("Ready...Set...Go!");

    PlayPingPongThread ping =
      new PlayPingPongThread("Ping!");
    PlayPingPongThread pong =
      new PlayPingPongThread("Pong!");

    ping.start();
    pong.start();

    ping.join();
    pong.join();

    System.out.println("Done!");
  ...
```

Create ping & pong threads

Implementation of the Buggy Ping-Pong Program

- Implementation of the PingPongWrong class

```
public class PingPongWrong {
  ...
  public static void main(String[] args) {
    ...
    System.out.println("Ready...Set...Go!");

    PlayPingPongThread ping =
      new PlayPingPongThread("Ping!");
    PlayPingPongThread pong =
      new PlayPingPongThread("Pong!");

    ping.start();
    pong.start();

    ping.join();
    pong.join();

    System.out.println("Done!");
  ...
```

Starts the ping & pong threads, which triggers their run() hook method

Implementation of the Buggy Ping-Pong Program

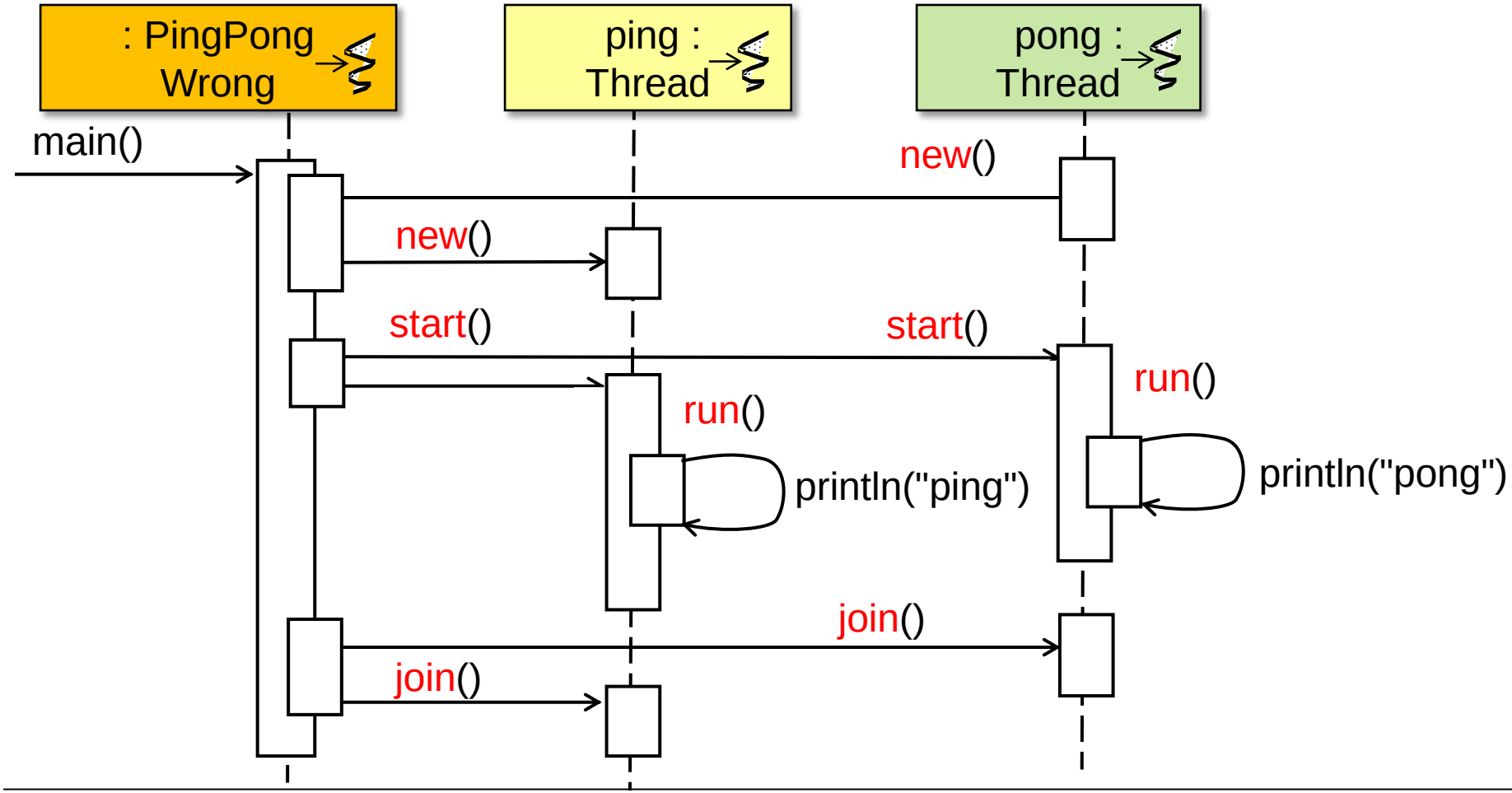
- Implementation of the PingPongWrong class

```
public class PingPongWrong {
  ...
  public static void main(String[] args) {
    ...
    System.out.println("Ready...Set...Go!");

    PlayPingPongThread ping =
      new PlayPingPongThread("Ping!");
    PlayPingPongThread pong =
      new PlayPingPongThread("Pong!");

    ping.start();
    pong.start();

    ping.join();
    pong.join();

    System.out.println("Done!");
  ...
```

Wait for both threads to exit

Implementation of the Buggy Ping-Pong Program

- This simple example showed how to create, start, run, & join Java threads



Evaluating the Buggy Ping-Pong Program

- This simple example showed how to create, start, run, & join Java threads
- The results are incorrect since there's no synchronization!

```
% java PingPongWrong
Ready...Set...Go!
Ping!(1)
Ping!(2)
Ping!(3)
Ping!(4)
Ping!(5)
Ping!(6)
Ping!(7)
Ping!(8)
Ping!(9)
Ping!(10)
Pong!(1)
Pong!(2)
Pong!(3)
Pong!(4)
Pong!(5)
Pong!(6)
Pong!(7)
Pong!(8)
Pong!(9)
Pong!(10)
Done!
```

Evaluating the Buggy Ping-Pong Program

- This simple example showed how to create, start, run, & join Java threads
- The results are incorrect since there's no synchronization!

```
% java PingPongWrong
Ready...Set...Go!
Ping!(1)
Ping!(2)
Ping!(3)
Ping!(4)
Ping!(5)
Ping!(6)
Ping!(7)
Ping!(8)
Ping!(9)
Ping!(10)
Pong!(1)
Pong!(2)
Pong!(3)
Pong!(4)
Pong!(5)
Pong!(6)
Pong!(7)
Pong!(8)
Pong!(9)
Pong!(10)
Done!
```

Evaluating the Buggy Ping-Pong Program

- This simple example showed how to create, start, run, & join Java threads
- The results are incorrect since there's no synchronization!

```
% java PingPongWrong
Ready...Set...Go!
Ping!(1)
Ping!(2)
Ping!(3)
Ping!(4)
Ping!(5)
Ping!(6)
Ping!(7)
Ping!(8)
Ping!(9)
Ping!(10)
Pong!(1)
Pong!(2)
Pong!(3)
Pong!(4)
Pong!(5)
Pong!(6)
Pong!(7)
Pong!(8)
Pong!(9)
Pong!(10)
Done!
```

Evaluating the Buggy Ping-Pong Program

- This simple example showed how to create, start, run, & join Java threads
- The results are incorrect since there's no synchronization
- We'll fix synchronization problem & produce the correct output after we cover Java Semaphores & CountDownLatches

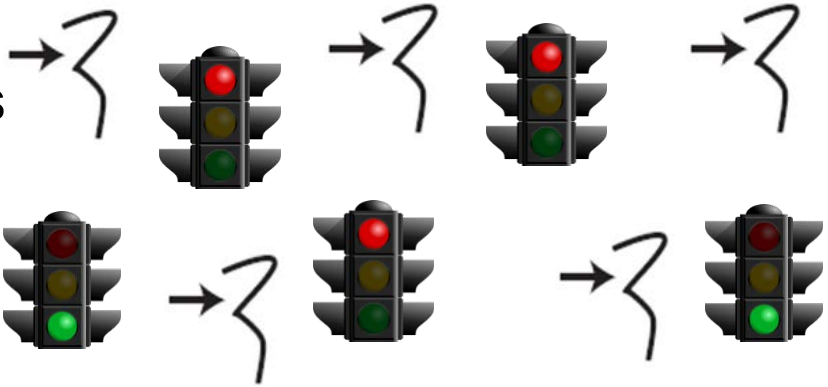
```
% java PingPong
Ready...Set...Go!
Ping!(1)
Pong!(1)
Ping!(2)
Pong!(2)
Ping!(3)
Pong!(3)
Ping!(4)
Pong!(4)
Ping!(5)
Pong!(5)
Ping!(6)
Pong!(6)
Ping!(7)
Pong!(7)
Ping!(8)
Pong!(8)
Ping!(9)
Pong!(9)
Ping!(10)
Pong!(10)
Done!
```

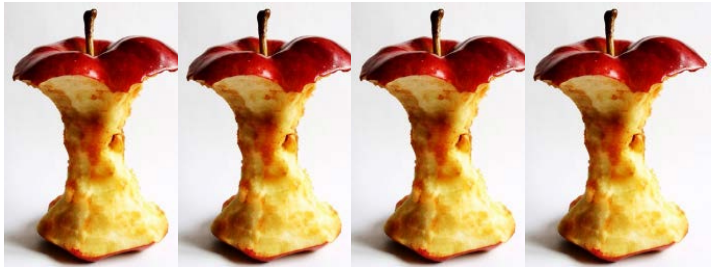
See upcoming part on "Java Synchronization & Scheduling Example"

Summary

Summary

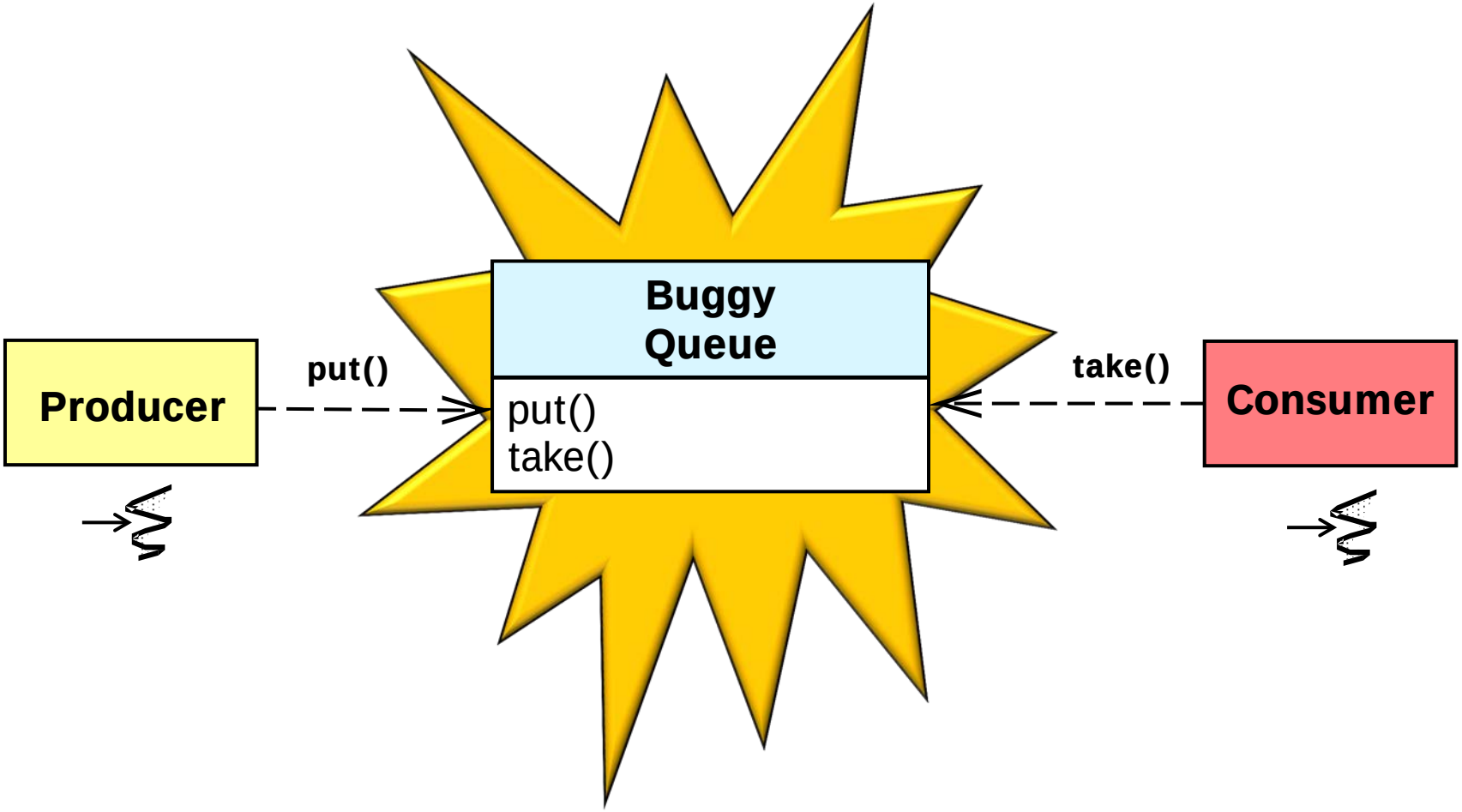
- Multi-threaded Java programs need synchronization & scheduling mechanisms





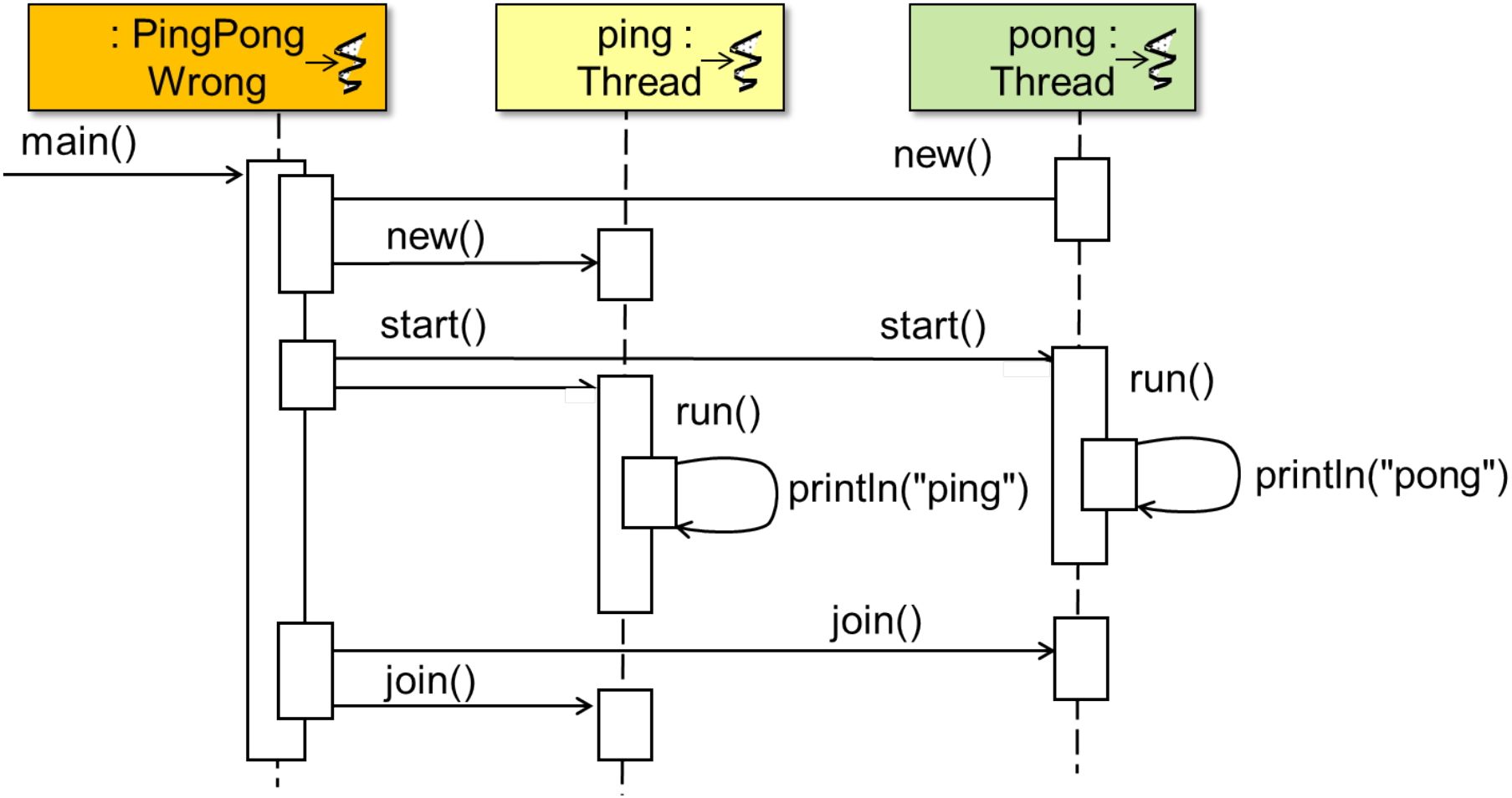
Summary

- Multi-threaded Java programs need synchronization & scheduling mechanisms



Summary

- Multi-threaded Java programs need synchronization & scheduling mechanisms



Summary

- Multi-threaded Java programs need synchronization & scheduling mechanisms
  - Prevent race conditions from corrupting shared data

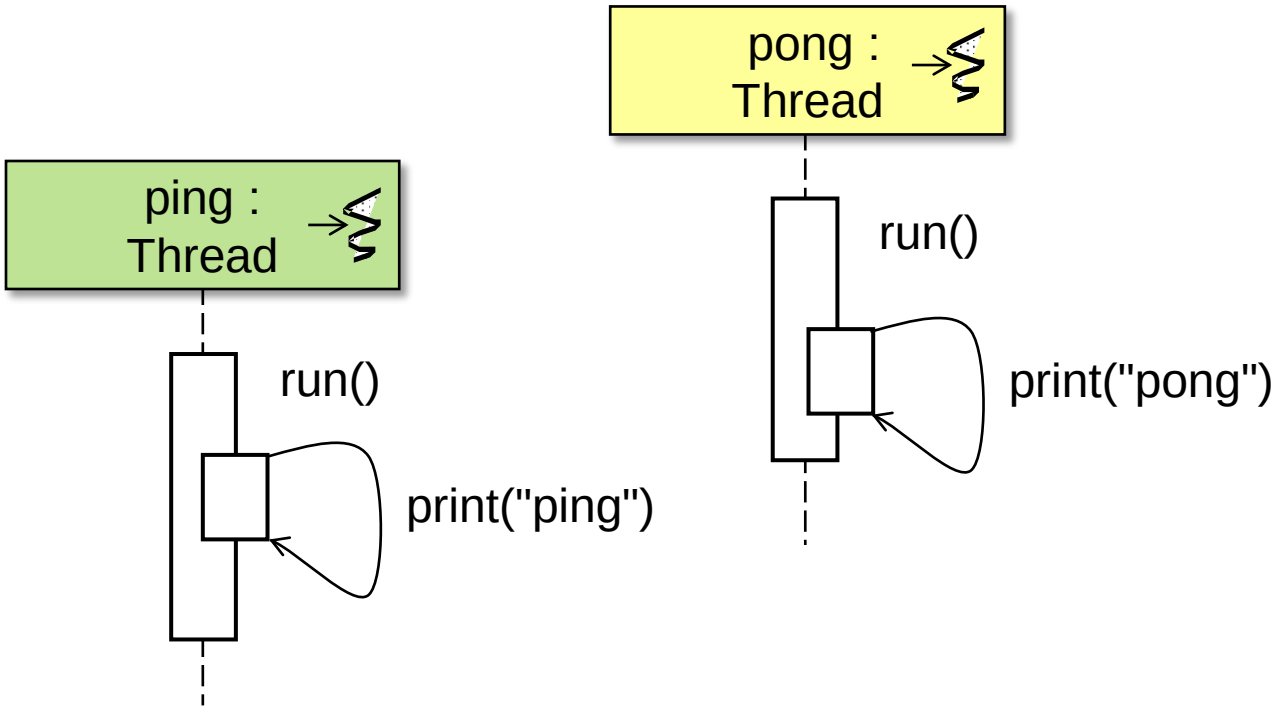
```
static class BuggyQueue {
  private List<String> mQ = new ArrayList<String>();


  public void put(String msg){ mQ.add(msg); }
```

There's no protection against critical sections being run by multiple threads concurrently

```
  public String take(){ return mQ.remove(0); }
}
```

Summary

- Multi-threaded Java programs need synchronization & scheduling mechanisms
  - Prevent race conditions from corrupting shared data
  - Ensure threads execute in the right order




```
% java PingPong
Ready...Set...Go!
Ping!(1)
Pong!(1)
Ping!(2)
Pong!(2)
Ping!(3)
Pong!(3)
Ping!(4)
Pong!(4)
Ping!(5)
Pong!(5)
Ping!(6)
Pong!(6)
Ping!(7)
Pong!(7)
Ping!(8)
Pong!(8)
Ping!(9)
Pong!(9)
Ping!(10)
Pong!(10)
Done!
```

Summary

- Multi-threaded Java programs need synchronization & scheduling mechanisms
- Both solutions are also hard-coded to work only with Java console applications

```
public class PingPongWrong {
    . . .
    public static class PlayPingPongThread
                                extends Thread {

        . . .
        public void run() {
            for (int loopsDone = 1;
                loopsDone <= mMaxIter;
                ++loopsDone)
                System.out.println
                (mStringToPri
```

Summary

- Multi-threaded Java programs need synchronization & scheduling mechanisms
- Both solutions are also hard-coded to work only with Java console applications
- We'll apply various "Gang-of-Four" & POSA patterns to make these solutions more portable, reusable, & extensible