Лекция 13

Многонишково Програмиране

Основни теми

- Какво представляват нишките и за какво се използват.
- Как нишките позволяват да се извършва съпътстваща (паралелна) обработка.
- Етапите в изпълнението на една нишка.
- Приоритети в изпълнението на нишки и програмиране.
- Създаване и изпълнение на `Runnable` обекти.

Основни теми

- Синхронизация на нишки.
- Задачата за **`Producer/ Consumer`** и нейното решение с многонишково програмиране.
- Многонишково програмиране и коректна **`(thread-safe`**) работа със **`Swing GUI`** компоненти.
- Използване на интерфейси **`Callable`** и **`Future`**, позволяващи нишка да връща данни в резултат от изпълнението си.

Задачи

Литература:

Java How to Program, 7 Edition, глава 23

23.1 Въведение

- **Известно е, че човек може да изпълнява много дейности едновременно- паралелно или съпътстващо с други (ходим и говорим, гледаме и пишем)**
- **Компютрите също могат да изпълняват операции паралелно (*parallel*) или съпътстващо (*concurrent*) с други**
- **Компютрите с един процесор изпълняват множество от операции съпътстващо (*concurrent*) , а тези с повече от един процесор изпълняват множество от операции паралелно (*parallel*)**

23.1 Въведение

- **ОС на компютрите с един процесор създават илюзия за едновременно изпълнение на множество задачи, чрез бързо превключване на процесора между контекстите на тези задачи, но в крайна сметка** ***във всеки даден момент процесора изпълнява точно една инструкция*** **принадлежаща на някоя от тези задачи.**
- ***Редът за превключване на процесора*** **между контекстите на изпълняваните задачи се нарича** ***Управление на заданията*** **на процесора и не може да се задава от потребителя на едно задание.**

23.1 Въведение

- Повечето програмни езици не дават възможност за паралелна обработка на задачи
- Преди години това изискваше експертни познания за програмиране на ниско ниво на ОС
- Езикът Ada е първият език, който включва средства за за едновременна обработка на задачи за целите на военната индустрия (системи за наблюдение и управление)
  - Строго ограничено засекретено приложение
- Java прави достъпно многонишково програмиране, чрез пакет от приложния програмен интерфейс (API)

23.1 Въведение

- **Приложение на едновременно изпълнение на операции (нишки) като част от задача**
  - **Изпълнение на аудио клип докато той се изтегля от Интернет**
  - **Изисква синхронизиране (координиране на действията) на нишките, така че нишката изпълняваща озвучаването да не започне преди достатъчно голяма част от клипа не е изтеглен от Интернет , а също на реагира при закъснения при изтегяне на файла**
- **Виртуалната машина на Java(JVM) създава нишки при изпълнение на всяка програма, JVM използва и допълнителни нишки за обслужване на изпълнението на програмата(“*събиране на боклук*”, *управление на събитията* в графичния интерфейс и др.)**

23.1 Въведение

- **Повечето съвременни ОС подържат изпълнението на задания като съвкупност от няколко операции**
- **Всяка *отделно дефинирана операция* като част от изпълнението на дадена програмна задача се нарича нишка**
- **Всяка програма започва изпълнението си като една *основна* нишка. В процеса на изпълнението й може да се дефинират *отделни операции* (нишки) и те да стартират независимото си изпълнение в рамките на програмата (разклонение на изпълнението на *основната* нишка)**

23.1 Въведение

Писането на многонишкова програма изисква повече усилия.

Да допуснем, че за подготовката си за изпит развивате даден въпрос като четете от две книги. Прочитате пасаж от първата книга, прехвърляте се на втората, записвате прочетеното и пак се връщате към една от двете книги.

***При това извършвате*:**

- **преминаване от една книга към друга**
- **преминаване от книга към записване**
- **запомняне къде и докъде е четено и записвано за последните няколко операции**

23.1 Въведение

- **Многонишковото програмиране изисква повече усилия за избягване на логически грешки**
  - **Някои общи правила**
    - ***Използвайте thread- safe класове от библиотеката на Java API*** **като например, class** `ArrayBlockingQueue` ***за гарантиране на синхронизация . Тези класове са тествани и оптимизирани за използваните ресурси на ОС.***
    - **За реализация на нишки в потребителски класове се използват редица конструкции (класове и интерфейси) на езика и ключови думи – например** `synchronized`**, методите** `wait`**,** `notify` **и** `notifyAll` **на клас** `Object`
    - **Допълнително, за реализация на условия при синхронизацията на нишките може да се използват интерфейси като** `Lock` **и** `Condition`

Съвет за по- добро качество 23.1

Програма реализирана като една нишка може да доведе до дълго чакане при взамодействи ес потребителя – по-продължителните операции трябва да приключат преди да се изпълни някоя с по- кратко време за изполнение. В многонишково задание нишките се изпълняват върху отделни процесори и отделните операции се изпълняват в паралел и ресурсите на компютъра с еизползват по-ефективно. Многонишковото програмиране повишава производителността на едно процесорните машини (с изключение на входно изходните операции).

23.2 Състояния на нишка: Етапи в изпълнение на нишка

- **Всяка нишка се намира е едно от следните състояния (Fig. 23.1)**
- **Нова нишка преди започване на изпълнение е в състояние *new*.**
- **При започване на изпълнението си преминава в състояние *runnable*.**
- ***Всяка runnable* нишка преминава в състояние *waiting* докато чака друга нишка да се изпълни (на процесора)**
  - **Преход обратно в състояние *runnable* става когато друга нишка изпрати сигнал на *waiting* нишката да продължи изпълнение.**

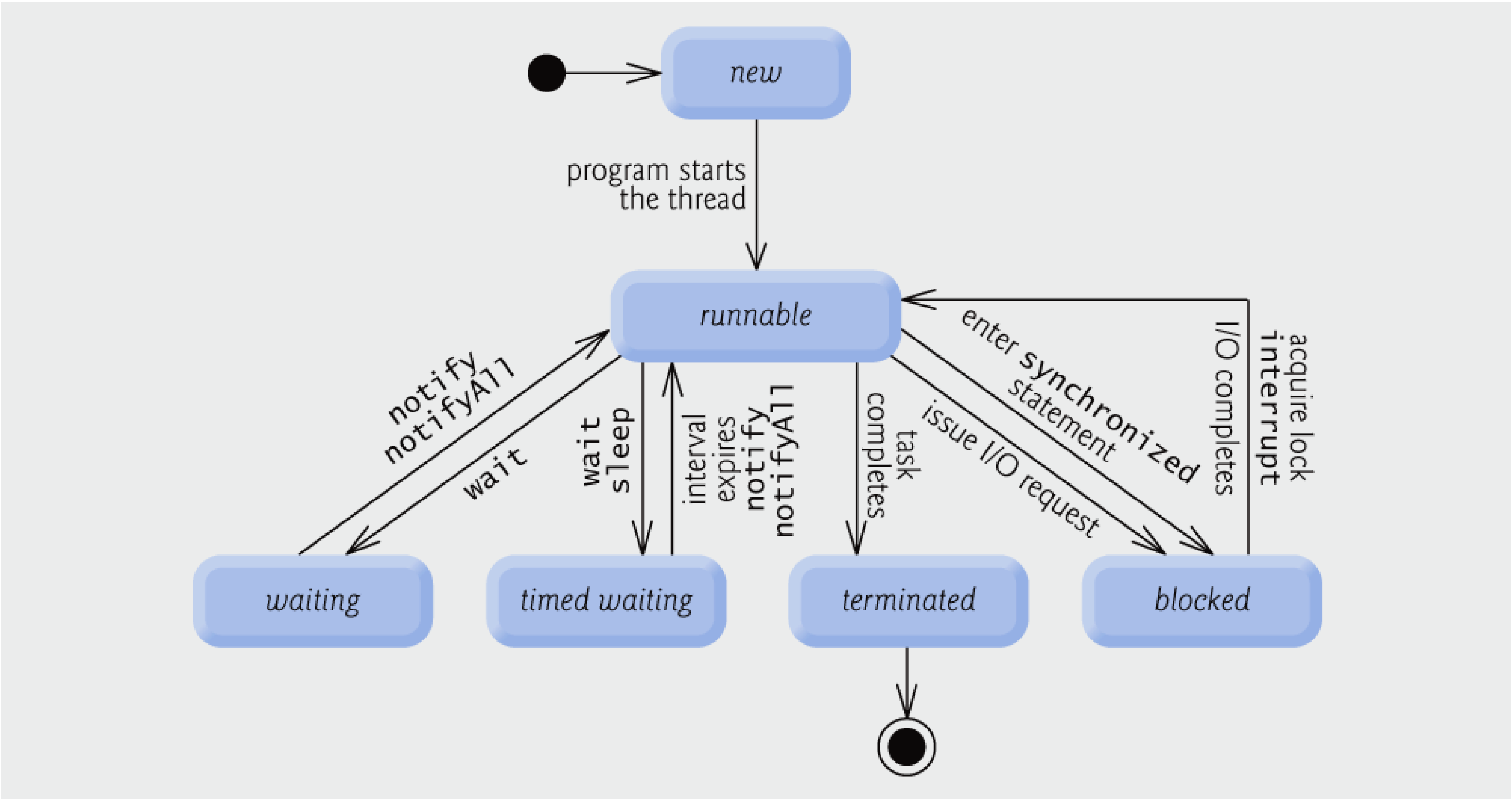


Fig. 23.1 | Основни състояния при изпълнение на нишка в Java.

23.2 Състояния на нишка: Етапи в изпълнение на нишка

- **Една *runnable* нишка може да премине в *timed waiting* състояние за предефиниран период от време или до настъпване на определено събитие**
  - **Преход обратно в *runnable* състояние се извършва когато зададеното време изтече или събитие , за което се изчаква настъпи**
- ***Timed waiting* и *waiting* нишки не използват процесорно време, дори, ако процесора е свободен да изпълнява задачи.**
- **Една *runnable* нишка може да преминава в *timed waiting* състояние, ако зададе по избор и параметър за време на чакане на друга нишка да приключи изпълнение .**
  - **Връща се в *runnable* състояние когато**
    - **Известена от друга нишка за настъпване на събитие или**
    - **Интервала на времето за чакане изтече**
- **Една нишка преминава в състояние на *timed waiting* когато е "*приспана*"**
  - **Остава в *timed waiting* състояние за зададен пероид от време и после се връща в *runnable* състояние**

23.2 Състояния на нишка: Етапи в изпълнение на нишка

- **Една *runnable* нишка преминава в *blocked* състояние, когато текущо изпълнявана от нея операция не може да се изпълни веднага и тя трябва да чака докато тази операция е възможно да се поднови- обикновено *входно изходни операции* водят до попадане в това състояние.**
  - **Една *blocked* нишка не може да използва процесора, дори да е свободен**
- **Една *runnable* нишка преминава в *terminated* състояние (още наричано *dead* състояние) ако е приключила изпълнението си – успешно или неуспешно (*вероятно в резултат на грешка*).**

23.2 Състояния на нишка: Етапи в изпълнение на нишка

- **На ниво Операционна система, състоянието *runnable* нишка в Java обикновено обхваща две състояния (Fig. 23.2).**
  - **ОС скрива тези състояния от JVM**
  - **Една *runnable* нишка първо влиза в *ready* състояние**
  - **Когато нишката се разпредели за изпълнение от процесора , то тя преминава в *running* състояние**
  - **Всички нишки получават част (*quantum*) от времето отредено за дадена задача за изпълнението й на процесора**
  - **Когато *quantum* за нишката изтече, нишката преминава отново в *ready* състояние , а ОС разпределя за изпълнение друга нишка**
  - **Прехода от *ready* в *runnable* и обратно се управлява от изцяло и единствено от ОС**

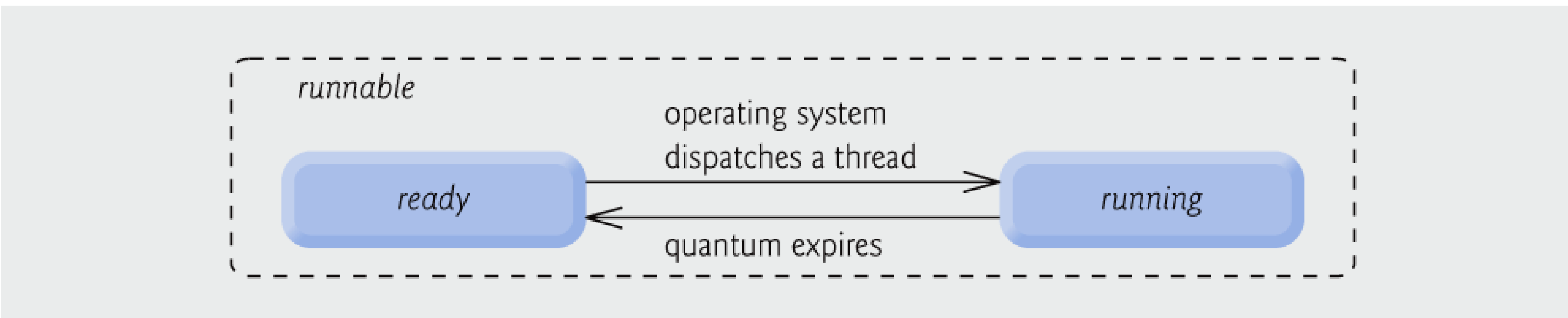


Fig. 23.2 | Представа за *runnable* състояние на ниво ОС.

23.3 Приоритети на нишка и изпълнение на нишка (`Thread`) от процесора

- **Всяка нишка на Java има thread приоритет, който позволява на ОС да *определи* (но не гарантира!) редът за изпълнение на нишките**
- **Приоритетите са в интервал от MIN_PRIORITY (константата 1) до MAX_PRIORITY (константата 10)**
- **По подразбиране на всяка нишка се дава NORM_PRIORITY (константата 5)**
- **Всяка нова нишка наследява приоритета на нишката, която я създава.**

23.3 Приоритети на нишка и изпълнение на нишка (`Thread`) от процесора

- **По правило, нишките с по- висок приоритет са по- важни за изпълнението на задачата и следва да използват процесора преди нишките с по- нисък приоритет**
- **Режим на Времеделене на работа на ОС**
  - **Позволява на нишки с еднакъв приоритет да се редуват при използване на процесора за част от време (*quantum*)**
  - **Когато *quantum* изтече процесора се разпределя на друга нишка с равен приоритет (ако има такава)**
- **Програмата на ОС за разпределение на нишките определя коя е следващата нишка за изпълнение**
- **Нишките с по- висок приоритет прекъсват (preempt) текущо изпълняваната нишка , ако е с по- нисък приоритет**
  - **Алгоритъм известен като *preemptive scheduling***
  - ***Проблем*: възможно е продължително отлагане на изпълнението на нишки с по- нисък приоритет**

Съвет за преносимост 23.1

Разпределението на нишки за изпълнение зависи от ОС и поведението на многонишковата програма на Java може да е различно за различно като производителност в различните ОС среди.

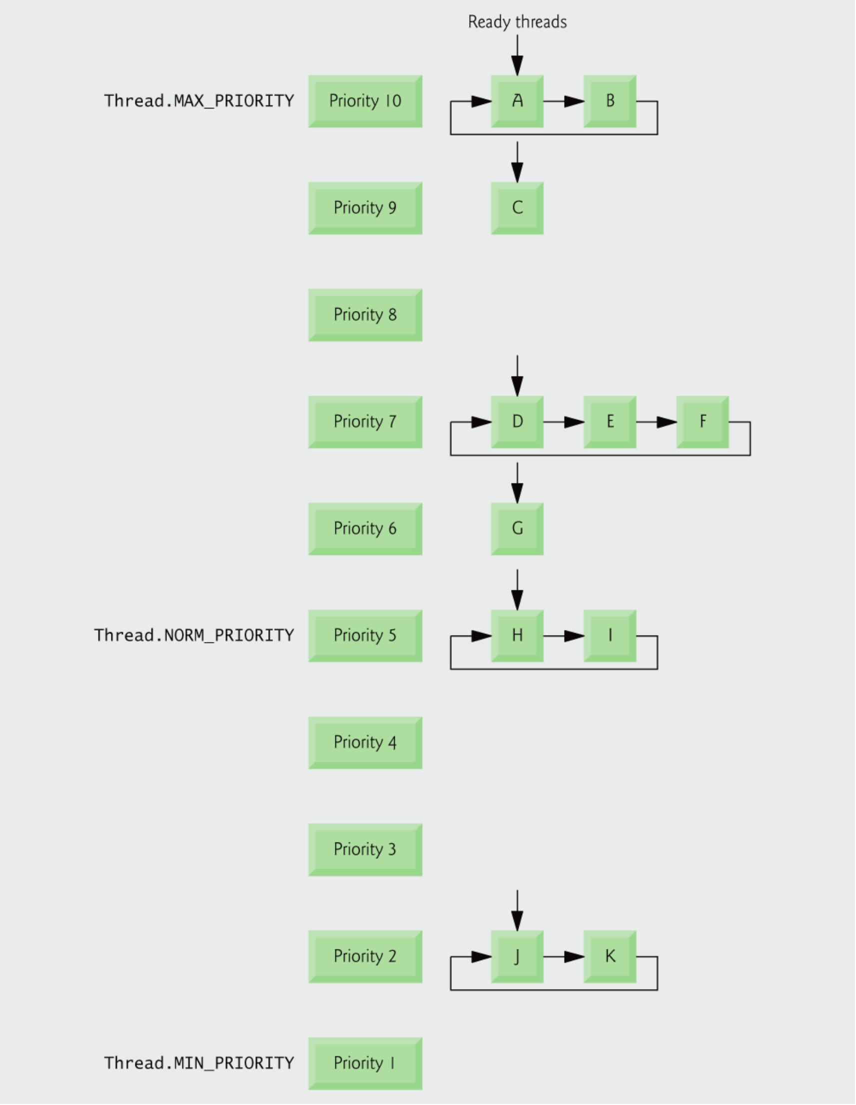


Fig. 23.3 | Разпределение на нишки за изпълнение с приоритети.

Съвет за преносимост 23.2

При моделиране на многонишкови програми с цел изпълнение на различни платформи, използването на приоритети различни от този по подразбиране може да доведе до големи различи в производителността на програмата.

23.4 Създаване и изпълнение на нишки

- След J2SE 5.0, предпочитаният начин за реализиране на многонишково приложение е да се имплементира `interface Runnable` (package `java.lang`) в класа, който ще дефинира метод (операция) за изпълнение в нишка
- `interface Runnable` декларира единствен метод `run`, който задава операцията изпълнявана от нишката
- Нишките изпълняват операции, дефинирани в `Runnable` обекти

23.4 Създаване и изпълнение на нишки

- Всяка нишка е обект от `class Thread`, чиито конструктор приема `Runnable` обект
- `Runnable` обекта задава "`операцията`", която да се изпълни едновременно с други операции
- Методът `run` на интерфейс `Runnable` дефинира командите за изпълнение в **нишка**

23.4.1 Интерфейс `Runnable` и Клас `Thread`

- **Пример:**

  Клас `PrintTask` (Fig. 23.4) имплементира `Runnable` (ред 5), така че всеки `PrintTask` обект да може да се изпълнява в отделна нишка.

  Променливата `sleepTime` (ред 7) генерира случайно число за "спане" (ред 17) дефинирано в конструктора на `PrintTask`.

  Така всяка нишка, изпълняваща `PrintTask` обект , спи за времето дефинирано в конструктора на `PrintTask` обекта и после извежда името си на печат

23.4.1 Интерфейс Runnable и Клас Thread

- **При създаване на нишката като Thread обект тя преминава в New състояние**
- **За преминаване в състояние Runnable нишката, която е Thread обект трябва за изпълни метода start() на class Thread**

23.4.1 Интерфейс `Runnable` и Клас `Thread`

- **Методът `sleep` хвърля изключение (checked) `InterruptedException` когато друга нишка изпълни метода `interrupt` на “*спящата*” нишка и така нишка се събужда и преминава отново в състояние `Runnable`**
- **Командите на метода `main` се изпълняват в основната нишка, която поражда други нишки посредством JVM**

Резюме

PrintTask.java

(1 от 2)

```
// Fig. 23.4: PrintTask.java
// PrintTask class sleeps for a random time from 0 to 5 seconds
import java.util.Random;

public class PrintTask implements Runnable
{
   private final int sleepTime; // random sleep time for thread
   private final String taskName; // name of task
   private final static Random generator = new Random();

   public PrintTask( String name )
   {
      taskName = name; // set task name

      // pick random sleep time between 0 and 5 seconds
      sleepTime = generator.nextInt( 5000 ); // milliseconds
   } // end PrintTask constructor

```

Имплементира `Runnable` за дефиниране на операция и изпълнението ѝ в нишка

Резюме

```
   // method run contains the code that a thread will execute
   public void run()
   {
      try // put thread to sleep for sleepTime amount of time
      {
         System.out.printf( "%s going to sleep for %d milliseconds.\n",
            taskName, sleepTime );
         Thread.sleep( sleepTime ); // put thread to sleep
      } // end try
      catch ( InterruptedException exception )
      {
         System.out.printf( "%s %s\n", taskName,
            "terminated prematurely due to interruption" );
      } // end catch

      // print task name
      System.out.printf( "%s done sleeping\n", taskName );
   } // end method run
} // end class PrintTask
```

PrintTask.java

(2 от 2)

Дефинира командите за изпълнениев нишка в метода **run**

Резюме

ThreadCreator.java

(1 от 2)

```
// Fig. 23.5: ThreadCreator.java
// Creating and starting three threads to execute Runnables.
import java.lang.Thread;

public class ThreadCreator
{
   public static void main( String[] args )
   {
      System.out.println( "Creating threads" );

      // create each thread with a new targeted runnable
      Thread thread1 = new Thread( new PrintTask( "task1" ) );
      Thread thread2 = new Thread( new PrintTask( "task2" ) );
      Thread thread3 = new Thread( new PrintTask( "task3" ) );

      System.out.println( "Threads created, starting tasks." );

      // start threads and place in runnable state
      thread1.start(); // invokes task1's run method
      thread2.start(); // invokes task2's run method
      thread3.start(); // invokes task3's run method

      System.out.println( "Tasks started, main ends.\n" );
   } // end main
} // end class RunnableTester
```

Създава `Thread` обекти за изпълнение на всеки нов `Runnable` обект

Стартира `Thread` обектите – нишките преминават в Runnable състояние

Резюме

ThreadCreator .java

(2 от 2)

```
Creating threads
Threads created, starting tasks
Tasks started, main ends

task3 going to sleep for 491 milliseconds
task2 going to sleep for 71 milliseconds
task1 going to sleep for 3549 milliseconds
task2 done sleeping
task3 done sleeping
task1 done sleeping
```

```
Creating threads
Threads created, starting tasks
task1 going to sleep for 4666 milliseconds
task2 going to sleep for 48 milliseconds
task3 going to sleep for 3924 milliseconds
Tasks started, main ends

thread2 done sleeping
thread3 done sleeping
thread1 done sleeping
```

23.4.2 Управление на изпълнението на нишки с `Executor` система от класове и интерфейси

- Препоръчва се изпълнението на interface `Executor` за управлението на `Runnable` обекти
- Всеки `Executor` създава управлява множество от `Thread` обекти, готови да изпълняват `Runnable` обекти
- Предимства от използване на `Executor`
  - Повторно използане на нишките без да се губи време за създаването им наново
  - Подобрява производителността на процесора- оптимизира броя на необходимите нишки

23.4.2 Управление на изпълнението на нишки с `Executor` система от класове и интерфейси

- **Методът `execute` на `Executor` приема `Runnable` обект като аргумент**
  - **Задава този `Runnable` обект на някоя от свободните нишки от множеството (thread pool)**
  - **Ако няма свободна нишка, то се създава нова нишка или се чака нишка от множеството (thread pool) да приключи изпълнението си**

23.4.2 с Executor система от класове и интерфейси

- Interface `ExecutorService`
  - package `java.util.concurrent`
  - Производен на `Executor`
  - Декларира методи за управление на състоянието на `Executor` обекти
  - `Executor` обекти се създават от `static` методи (например execute()) на class `Executors` (package `java.util.concurrent`)
- Методът `newCachedThreadPool` на `Executors` връща `ExecutorService` обект, който може да създава нишки при необходимост
- Методът `execute` на `ExecutorService` връща веднага след изпълнението си
- Методът `shutdown` на `ExecutorService` известява `ExecutorService` да спре приемането на нови задания , но да изчака приключване на текущо изпълняваните и тогава за приключи изпълнението си.

Резюме

```java
// Fig. 23.6: TaskExecutor.java
// Using an ExecutorService to execute Runnables.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class TaskExecutor
{
   public static void main( String[] args )
   {
      // create and name each runnable
      PrintTask task1 = new PrintTask( "task1" );
      PrintTask task2 = new PrintTask( "task2" );
      PrintTask task3 = new PrintTask( "task3" );

      System.out.println( "Starting Executor" );

      // create ExecutorService to manage threads
      ExecutorService threadExecutor = Executors.newCachedThreadPool();

```

TaskExecutor.java

Създава `ExecutorService` за управляване на кеширано множество от нишки (thread pool)

```
      // start threads and place in runnable state
      threadExecutor.execute( task1 ); // start task1
      threadExecutor.execute( task2 ); // start task2
      threadExecutor.execute( task3 ); // start task3

      // shut down worker threads when their tasks complete
      threadExecutor.shutdown();

      System.out.println( "Tasks started, main ends.\n" );
   } // end main
} // end class TaskExecutor
```

Използва метод `execute` на `ExecutorService` за задаване на изпълнение на всеки нов `Runnable` обект като нишка от множество thread pool

`TaskExecutor .java`

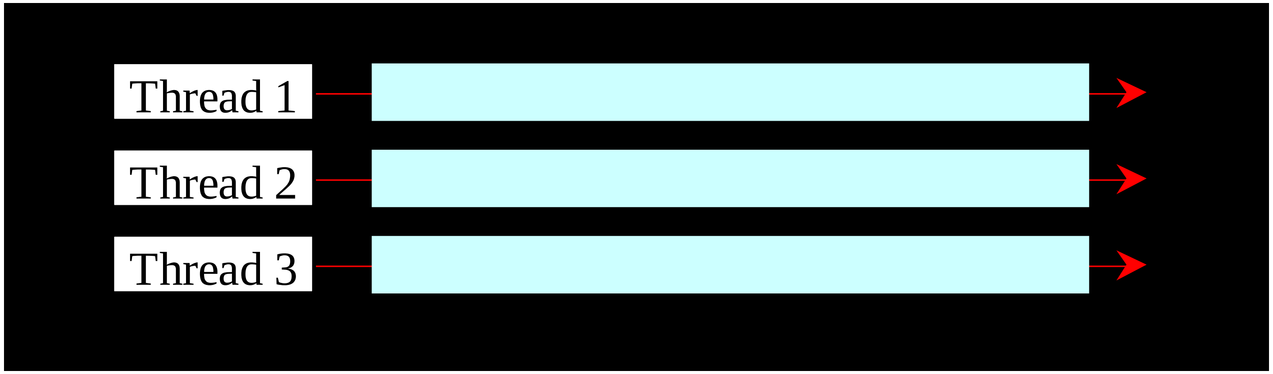
```
Starting Executor
Tasks started, main ends

task1 going to sleep for 4806 milliseconds
task2 going to sleep for 2513 milliseconds
task3 going to sleep for 1132 milliseconds
thread3 done sleeping
thread2 done sleeping
thread1 done sleeping
```

`ExecutorService` спира получаване на нови заявки за изпълнение на нишки чрез нови `Runnable` обекти

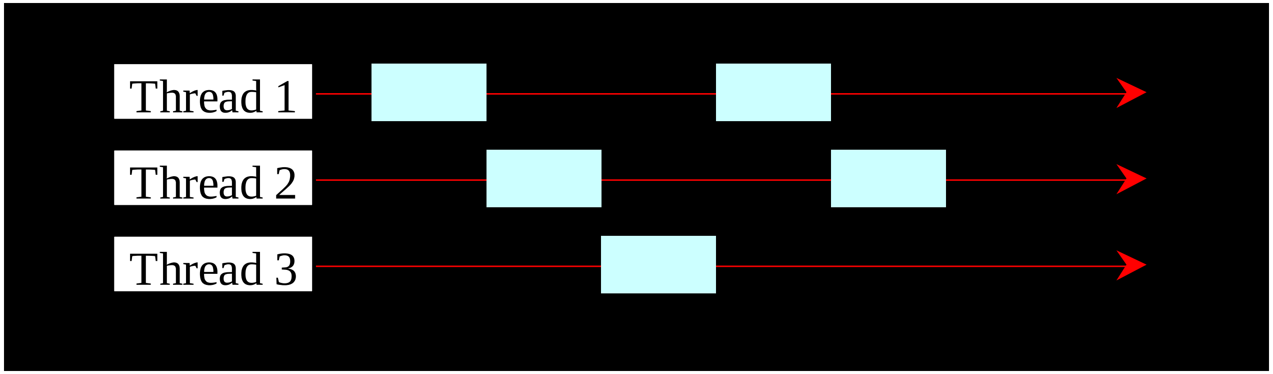
```
Starting Executor
task1 going to sleep for 1342 milliseconds
task2 going to sleep for 277 milliseconds
task3 going to sleep for 2737 milliseconds
Tasks started, main ends

task2 done sleeping
task1 done sleeping
task3 done sleeping
```

23.4 Обобщение

Нишки, изпълнявани на отделни CPU



Нишки, изпълнявани на едно CPU



23.4 Обобщение

java.lang.Runnable ← TaskClass

```java
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```java
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

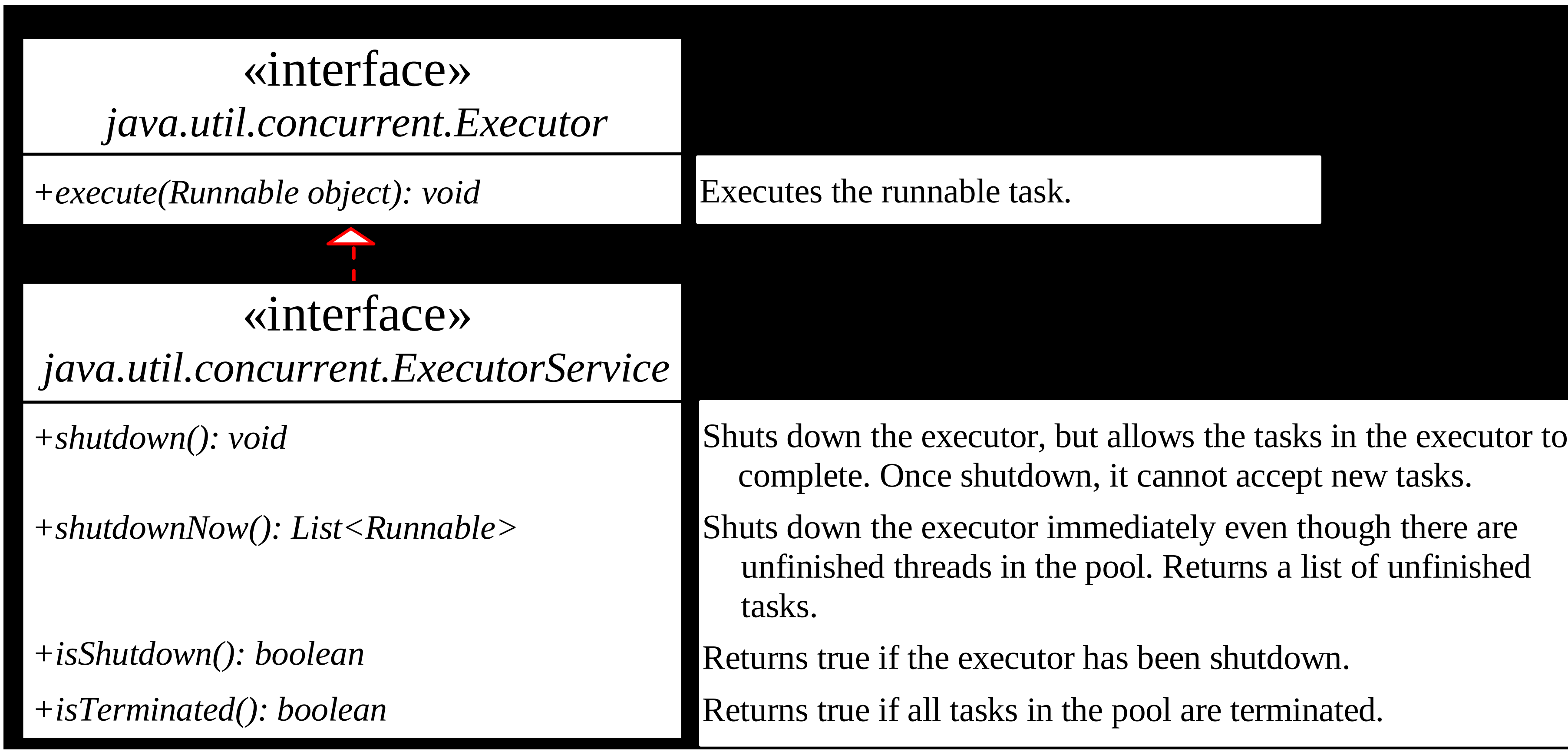
23.4.2 Управление на изпълнението на нишки с Executor система от класове и интерфейси

- **Thread pool (*множество нишки за многократно използване*)**
- **Стартиране на нова нишка за всяка нова операция може да влоши изпълнението и да намали производителността на програмата**
- **Поддържане на множество от готови за стартиране обекти- нишки. JDK 1.5 използва:**
  - **интерфейс `Executor` за изпълняване на операции посредством Thread pool**
  - **интерфейс `ExecutorService` за управление и обработка на операции**
  - **`ExecutorService` е производен интерфейс на `Executor`**

23.4.2 Управление на изпълнението на нишки с `Executor` система от класове и интерфейси

- **Thread pool видове**
  - **Cached (***създават се нови нишки, ако няма свободни, при завършване на операция се добавят към множеството от нишки, готови да обслужват други операции***)**
  - **Fixed (** *при създаването на Executor обект се създава фиксиран брой нишки, готови да обслужват операции и техният брой не се мени- ако има нужда да се изпълни операция, а няма свободна нишка в множеството се изчаква изпълнявана операция да приключи и се използва освободената от нея нишка***)**

23.4.2 Управление на изпълнението на нишки с Executor система от класове и интерфейси



23.4.2 Управление на изпълнението на нишки с Executor система от класове и интерфейси

- Препоръчва се изпълнението на interface `Executor` за управлението на `Runnable` обекти
- Всеки `Executor` създава управлява множество от `Thread` обекти, готови да изпълняват `Runnable` обекти
- Предимства от използване на `Executor`
  - Повторно използане на нишките без да се губи време за създаването им наново
  - Подобрява производителността на процесора- оптимизира броя на необходимите нишки

23.4.2 Управление на изпълнението на нишки с `Executor` система от класове и интерфейси

- **Методът `execute` на `Executor` приема `Runnable` обект като аргумент**
  - **Задава този `Runnable` обект на някоя от свободните нишки от множеството (thread pool)**
  - **Ако няма свободна нишка, то се създава нова нишка или се чака нишка от множеството (thread pool) да приключи изпълнението си**

23.4.2 c Executor система от класове и интерфейси

- Interface `ExecutorService`
  - package `java.util.concurrent`
  - Производен на `Executor`
  - Декларира методи за управление на състоянието на `Executor` обекти
  - `Executor` обекти се създават от `static` методи (например execute()) на class `Executors` (package `java.util.concurrent`)
- Методът `newCachedThreadPool` на `Executors` връща `ExecutorService` обект, който може да създава нишки при необходимост
- Методът `execute` на `ExecutorService` връща веднага след изпълнението си
- Методът `shutdown` на `ExecutorService` известява `ExecutorService` да спре приемането на нови задания , но да изчака приключване на текущо изпълняваните и тогава за приключи изпълнението си.

```java
// Fig. 23.6: TaskExecutor.java
// Using an ExecutorService to execute Runnables.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class TaskExecutor
{
   public static void main( String[] args )
   {
      // create and name each runnable
      PrintTask task1 = new PrintTask( "task1" );
      PrintTask task2 = new PrintTask( "task2" );
      PrintTask task3 = new PrintTask( "task3" );

      System.out.println( "Starting Executor" );

      // create ExecutorService to manage threads
      ExecutorService threadExecutor = Executors.newCachedThreadPool();

```

TaskExecutor .java

(1 of 2)

Създава `ExecutorService` за управляване на кеширано множество от нишки (thread pool)

```java
      // start threads and place in runnable state
      threadExecutor.execute( task1 ); // start task1
      threadExecutor.execute( task2 ); // start task2
      threadExecutor.execute( task3 ); // start task3

      // shut down worker threads when their tasks complete
      threadExecutor.shutdown();

      System.out.println( "Tasks started, main ends.\n" );
   } // end main
} // end class TaskExecutor
```

Използва метод `execute` на `ExecutorService` за задаване на изпълнение на всеки нов `Runnable` обект като нишка от множество thread pool

TaskExecutor .java

```
Starting Executor
Tasks started, main ends

task1 going to sleep for 4806 milliseconds
task2 going to sleep for 2513 milliseconds
task3 going to sleep for 1132 milliseconds
thread3 done sleeping
thread2 done sleeping
thread1 done sleeping
```

`ExecutorService` спира получаване на нови заявки за изпълнение на нишки чрез нови `Runnable` обекти

```
Starting Executor
task1 going to sleep for 1342 milliseconds
task2 going to sleep for 277 milliseconds
task3 going to sleep for 2737 milliseconds
Tasks started, main ends

task2 done sleeping
task1 done sleeping
task3 done sleeping
```

23.5 Синхронизация на нишки

- Задача: Да се координира достъпа на множество от едновременно изпълнявани нишки до данна споделена от тези нишки
  - При отсъствие на синхронизиран достъп да споделени данни се получават неопределени резултати ри изпълнение на програма
  - Данна споделена между няколко нишки остава в неопределено състояние при едновременен достъп от няколко нишки

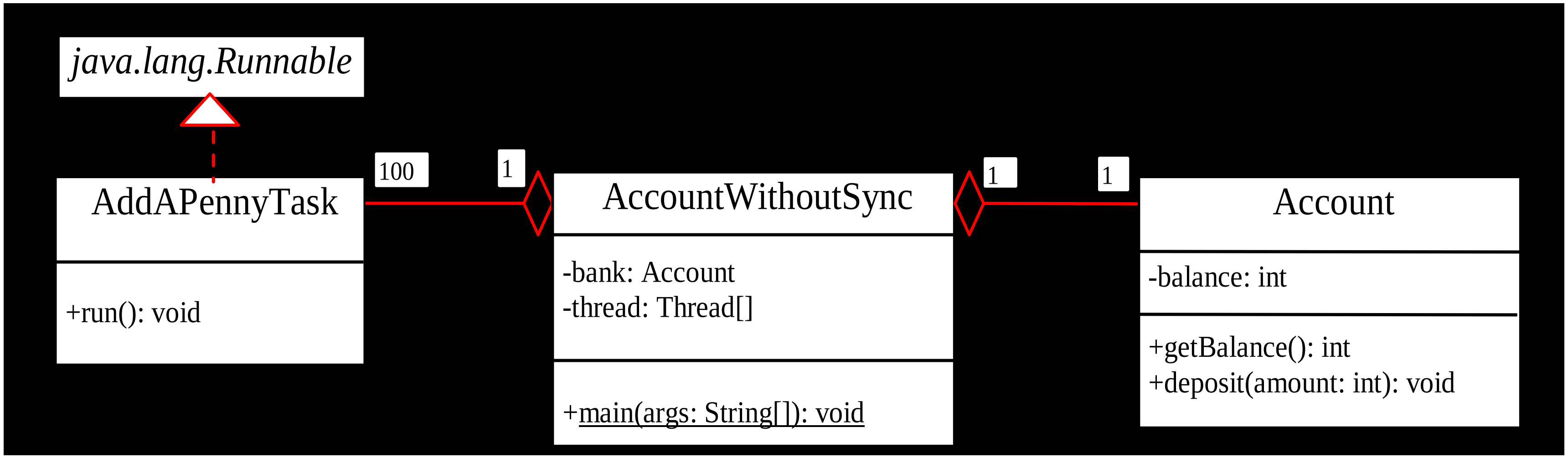
Пример: Две нишки обработващи една и съща банкова сметка

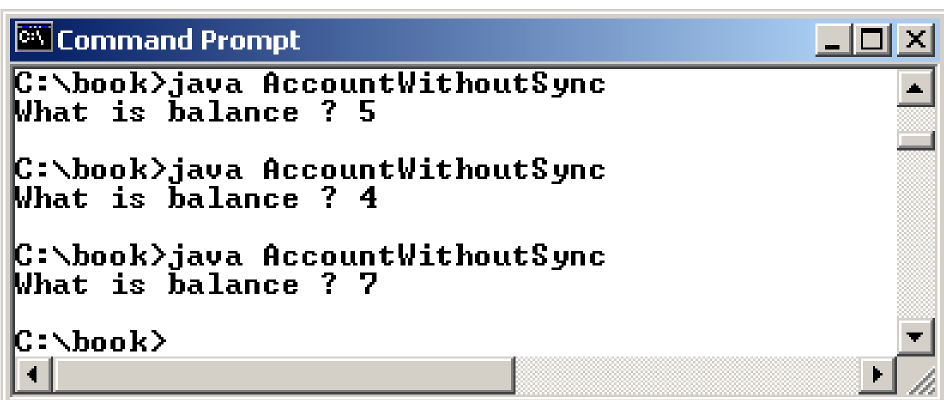
23.5 Синхронизация на нишки

- **Да предположим, че 100 нишки изпълняват операция с дадена банкова сметка, при което към банковата сметка се добавя по стотинка. Нека в началото банковата сметка да е празна**
- **След изпълнението си всяка от нишките проверява баланса на банковата сметка. Оказва се ,че балансът е с неопределена стойност.**

Step	balance	thread[i]	thread[j]
1	0	newBalance = bank.getBalance() + 1;	
2	0		newBalance = bank.getBalance() + 1;
3	1	bank.setBalance(newBalance);	
4	1		bank.setBalance(newBalance);

23.5 Синхронизация на нишки





23.5 Синхронизация на нишки

- **Race Conditions (***условия за надпревара между нишките за използване на процесора във всеки един момент водят до неопределен резултат в обработването на споделена данна***)**

Step	balance	Task 1	Task 2
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;

`

23.5 Синхронизация на нишки

Заради *лииса на синхронизация*, ефектът от изпълнение на Task 2 е нулев, понеже на Спъпка 4 балансът променен с Task 2 се изтрива с резултатът от Task 1. Проблемът е, че Task 1 и Task 2 имат конфликт при достъп до данната баланс. Този проблем се нарича *race condition* при многонишково програмиране.

Един клас се нарича *thread-safe* ако обект от този клас не се влияе *от* *race condition* при многонишков достъп до него. В дадения пример клас Account class не е *thread-safe*.

23.5 Синхронизация на нишки

- **Задача: Да се координира достъпа на множество от едновременно изпълнявани нишки до данна споделена от тези нишки**
  - **При отсъствие на синхронизиран достъп да споделени данни се получават неопределени резултати ри изпълнение на програма**

23.5 Синхронизация на нишки

Решение:

- Да се разреши **изключителен достъп** до споделената данна **на една единствена нишка** във всеки отделен момент
- През това време останалите нишки, изискващи достъп до данната преминават в състояние **WAITING** (изчакват на опашка)
- Когато нишката с изключителен достъп до данната приключи работа с данната, **една от изчакващите на опашка нишки получава достъп** до данната

- **Взаимно изключващ** достъп до данната

23.5 Синхронизация на нишки

- **Java използва вградени “наблюдател”-и (“monitor”) за реолизиране на синхронизация на достъп до данни**
- **Всеки обект има *monitor* и *monitor lock (ключ)***
  - **Мониторът осигурява достъп до ключа на обекта до една единствена нишка във всеки отделен момент**
  - **Използва се за реализиране на взаимно изключван достъп до споделяна данна**
- **За налагане на *взаимно изключване***
  - **Една нишка трябва да получи достъп до ключа за обекта**
  - **Нишката заключва достъп до обекта , до приключване на работа с него**
  - **Други нишки, опитващи да извършане на операция с този обект изчакват ключа на обекта да стане свободен**

23.5 Синхронизация на нишки

- `synchronized` команда
  - Задава взаимно изключван достъп до данна
  - ```
    synchronized ( object )
    {
        statements
    } // end synchronized statement
    ```
  - Където *object* е обектът, чиито ключ ще е необходим преди получаване на достъп до обекта)
- Един `synchronized` метод е еквивалентен на `synchronized` команда , обхващаща тялото на метода
    ```

23.5.1 Пример за несинхронизиран достъп

- Методът `awaitTermination` на `ExecutorService` налага програмата да изчака всички нишки от Thread pool да приключат изпълнение
  - Връща управление на извикващата програма, когато всички изпълнявани операции от нишките на `ExecutorService` завършат или изтече зададено време
  - Ако операциите приключат преди граничното време `awaitTermination` връща `true`; в противен случай връща `false`

```
// Fig. 23.7: SimpleArray.java
// Class that manages an integer array to be shared by multiple threads.
import java.util.Random;

public class SimpleArray // CAUTION: NOT THREAD SAFE!
{
   private final int array[]; // the shared integer array
   private int writeIndex = 0; // index of next element to be written
   private final static Random generator = new Random();

   // construct a SimpleArray of a given size
   public SimpleArray( int size )
   {
      array = new int[ size ];
   } // end constructor

   // add a value to the shared array
   public void add( int value )
   {
      int position = writeIndex; // store the write index

      try
      {
         // put thread to sleep for 0-499 milliseconds
         Thread.sleep( generator.nextInt( 500 ) );
      } // end try
      catch ( InterruptedException ex )
      {
         ex.printStackTrace();
      } // end catch
```

Задава индексът за записване на следващия елемент

Текущата нишка спи 500 ms преди да запише стойност

Outline

SimpleArray.java

(1 от 2)

```java

        // put value in the appropriate element
        array[ position ] = value;
        System.out.printf( "%s wrote %2d to element %d.\n",
           Thread.currentThread().getName(), value, position );

        ++writeIndex; // increment index of element to be written next
        System.out.printf( "Next write index: %d\n", writeIndex );
     } // end method add

     // used for outputting the contents of the shared integer array
     public String toString()
     {
        String arrayString = "\nContents of SimpleArray:\n";

        for ( int i = 0; i < array.length; i++ )
           arrayString += array[ i ] + " ";

        return arrayString;
     } // end method toString
  } // end class SimpleArray
```

Добавя нова стойност

Задава индекс за записване на следващата стойност

SimpleArray.java

(2 от 2)

```java
// Fig. 23.8: ArrayWriter.java
// Adds integers to an array shared with other Runnables
import java.lang.Runnable;

public class ArrayWriter implements Runnable
{
   private final SimpleArray sharedSimpleArray;
   private final int startValue;

   public ArrayWriter( int value, SimpleArray array )
   {
      startValue = value;
      sharedSimpleArray = array;
   } // end constructor

   public void run()
   {
      for ( int i = startValue; i < startValue + 3; i++ )
      {
         sharedSimpleArray.add( i ); // add an element to the shared array
      } // end for
   } // end method run
} // end class ArrayWriter
```


ArrayWriter.java

Outline

SharedArrayTest.java

(1 от 2)

```
// Fig 23.9: SharedArrayTest.java
// Executes two Runnables to add elements to a shared SimpleArray.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

public class SharedArrayTest
{
   public static void main( String[] arg )
   {
      // construct the shared object
      SimpleArray sharedSimpleArray = new SimpleArray( 6 );

      // create two tasks to write to the shared SimpleArray
      ArrayWriter writer1 = new ArrayWriter( 1, sharedSimpleArray );
      ArrayWriter writer2 = new ArrayWriter( 11, sharedSimpleArray );

      // execute the tasks with an ExecutorService
      ExecutorService executor = Executors.newCachedThreadPool();
      executor.execute( writer1 );
      executor.execute( writer2 );

      executor.shutdown();

      try
      {
         // wait 1 minute for both writers to finish executing
         boolean tasksEnded = executor.awaitTermination(
            1, TimeUnit.MINUTES );

```

Двата обекта `ArrayWriter` споделят **същия** `SimpleArray` обект

```
         if ( tasksEnded )
            System.out.println( sharedSimpleArray ); // print contents
         else
            System.out.println(
               "Timed out while waiting for tasks to finish." );
      } // end try
      catch ( InterruptedException ex )
      {
         System.out.println(
            "Interrupted while wait for tasks to finish." );
      } // end catch
   } // end main
} // end class SharedArrayTest
```

```
pool-1-thread-1 wrote  1 to element 0.
Next write index: 1
pool-1-thread-1 wrote  2 to element 1.
Next write index: 2
pool-1-thread-1 wrote  3 to element 2.
Next write index: 3
pool-1-thread-1 wrote 11 to element 0.
Next write index: 4
pool-1-thread-2 wrote 12 to element 4.
Next write index: 5
pool-1-thread-2 wrote 13 to element 5.
Next write index: 6

Contents of SimpleArray:
11 2 3 0 12 13
```

First `pool-1-thread-1` wrote the value 1 to element 0. Later `pool-1-thread-2` wrote the value 11 to element 0, thus overwriting the previously stored value.

23.5.2 Синхронизиране на данни

- **Симулираме неделимост на операцията като налагаме ограничението една единствена нишка да има достъп до споделената данна във всеки отделен момент**
- **Immutable данни могат да се споделят между нишки**
  - **Декларираме данните на обекта като `final` за да отбележим, че те не трябва да се променят след първоначалната им инициализация**

23.5.2 Синхронизиране на данни

- **The sleep method** ***"Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. The thread <u>does not lose ownership of any monitors</u>."*** ***(SUN)***

Software Engineering факт 23.1

Поставете всеки възможен достъп до изменяеми споделени данни между няколко нишки вътре в `synchronized` блокове или `synchronized` методи за да бъдат синхронизирани по отношение на един и същ ключ. При извършване на няколко операции с даден обект, пазете ключа по време на цялата операция, за да бъде тя изпълнена наистина неделима на процесора.

```java
// Fig. 23.10: SimpleArray.java
// Class that manages an integer array to be shared by multiple threads.
import java.util.Random;

public class SimpleArray
{
   private final int array[]; // the shared integer array
   private int writeIndex = 0; // index of next element to be written
   private final static Random generator = new Random();

   // construct a SimpleArray of a given size
   public SimpleArray( int size )
   {
      array = new int[ size ];
   } // end constructor

   // add a value to the shared array
   public synchronized void add( int value )
   {
      int position = writeIndex; // store the write index

      try
      {
         // put thread to sleep for 0-499 milliseconds
         Thread.sleep( generator.nextInt( 500 ) );
      } // end try
      catch ( InterruptedException ex )
      {
         ex.printStackTrace();
      } // end catch
```

Използване на `synchronized` **не допуска повече от една нишка да извика този метод** за даден `SimpleArray` обект

```
      // put value in the appropriate element
      array[ position ] = value;
      System.out.printf( "%s wrote %2d to element %d.\n",
         Thread.currentThread().getName(), value, position );

      ++writeIndex; // increment index of element to be written next
      System.out.printf( "Next write index: %d\n", writeIndex );
   } // end method add

   // used for outputting the contents of the shared integer array
   public String toString()
   {
      String arrayString = "\nContents of SimpleArray:\n";

      for ( int i = 0; i < array.length; i++ )
         arrayString += array[ i ] + " ";

      return arrayString;
   } // end method toString
} // end class SimpleArray
```


SimpleArray.java

(2 от 3)

```
pool-1-thread-1 wrote  1 to element 0.
Next write index: 1
pool-1-thread-2 wrote 11 to element 1.
Next write index: 2
pool-1-thread-2 wrote 12 to element 2.
Next write index: 3
pool-1-thread-2 wrote 13 to element 3.
Next write index: 4
pool-1-thread-1 wrote  2 to element 4.
Next write index: 5
pool-1-thread-1 wrote  3 to element 5.
Next write index: 6

Contents of SimpleArray:
1 11 12 13 2 3
```


SimpleArray.java

(3 от 3)

Съвет за добро качество 23.2

Ограничавайте използването на `synchronized` команди до възможно най-кратко време, необходимо за Това минимизира времето за чакане на блокираните нишки. Избягвайте използване на дълги I/O, пресмятания и други с тази команда, които не изискват синхронизация с ключ.

Добра практика на програмиране 23.1

Винаги декларирайте данните на обект, които няма да се променят като `final`. Примитивни данни, декларирани като `final` могат да се споделят без проблем между нишки. Референция към обект, декларирана като `final` гарантира, че този обект ще се създаде и инициализира преди да се използва и предпазва да се смени тази референция към друг обект

23.6 Задачата Производител-Потребител без синхронизация

- **Многонишково producer/consumer приложение**
  - **Producer нишка създава данни и ги записва последователно в буфер споделян с друга нишка**
  - **Consumer нишката чете последователно (по веднъж) от споделения буфер**
- **Операциите по обработка на буфера зависят от неговото състояние**
  - **Могат да се извършат, ако буферът е в подходящо състояние**
  - **Производителят може да работи ако буферът не е пълен**
  - **Потребителят може да работи ако буферът не е празен**
- **Трябва да се синхронизира достъпа, така че операциите да могат да извършват или не в зависимост от състоянието на буфера**

```java
// Fig. 23.11: Buffer.java
// Buffer interface specifies methods called by Producer and Consumer.
public interface Buffer
{
   // place int value into Buffer
   public void set( int value ) throws InterruptedException;

   // return int value from Buffer
   public int get() throws InterruptedException;
} // end interface Buffer
```

Outline

Buffer.java

Интерфейс `Buffer` ще използваме във следващите примери на тази задача

```java
// Fig. 23.12: Producer.java
// Producer with a run method that inserts the values 1 to 10 in buffer.
import java.util.Random;

public class Producer implements Runnable
{
   private final static Random generator = new Random();
   private final Buffer sharedLocation; // reference to shared object

   // constructor
   public Producer( Buffer shared )
   {
      sharedLocation = shared;
   } // end Producer constructor

   // store values from 1 to 10 in sharedLocation
   public void run()
   {
      int sum = 0;

      for ( int count = 1; count <= 10; count++ )
      {
         try // sleep 0 to 3 seconds, then place value in Buffer
         {
            Thread.sleep( generator.nextInt( 3000 ) ); // random sleep
            sharedLocation.set( count ); // set value in buffer
            sum += count; // increment sum of values
            System.out.printf( "\t%2d\n", sum );
         } // end try
```

Клас `Producer` е `Runnable` и записва данни в `Buffer`

Дефинира операцията на `Producer`

Записва стойност в `Buffer`

Outline

Producer.java

(1 от 2)

```java
            // if lines 25 or 26 get interrupted, print stack trace
            catch ( InterruptedException exception )
            {
               exception.printStackTrace();
            } // end catch
         } // end for

         System.out.println(
            "Producer done producing\nTerminating Producer" );
      } // end method run
} // end class Producer
```

Outline

Producer.java

(2 от 2)

```
// Fig. 23.13: Consumer.java
// Consumer with a run method that loops, reading 10 values from buffer.
import java.util.Random;

public class Consumer implements Runnable
{
   private final static Random generator = new Random();
   private final Buffer sharedLocation; // reference to shared object

   // constructor
   public Consumer( Buffer shared )
   {
      sharedLocation = shared;
   } // end Consumer constructor

   // read sharedLocation's value 10 times and sum the values
   public void run()
   {
      int sum = 0;

      for ( int count = 1; count <= 10; count++ )
      {
         // sleep 0 to 3 seconds, read value from buffer and add to sum
         try
         {
            Thread.sleep( generator.nextInt( 3000 ) );
            sum += sharedLocation.get();
            System.out.printf( "\t\t\t%2d\n", sum );
         } // end try
```

Клас `Consumer` е `Runnable` и чете данни от `Buffer`

Дефинира операцията на `Consumer`

Чете стойност от `Buffer`

Outline

Consumer.java

(1 от 2)

```java
         // if lines 26 or 27 get interrupted, print stack trace
         catch ( InterruptedException exception )
         {
            exception.printStackTrace();
         } // end catch
      } // end for

      System.out.printf( "\n%s %d\n%s\n",
         "Consumer read values totaling", sum, "Terminating Consumer" );
   } // end method run
} // end class Consumer
```

Outline

UnsynchronizedBuffer.java

```
// Fig. Fig. 23.14: UnsynchronizedBuffer.java
// UnsynchronizedBuffer maintains the shared integer that is accessed by
// a producer thread and a consumer thread via methods set and get.
public class UnsynchronizedBuffer implements Buffer
{
   private int buffer = -1; // shared by producer and consumer threads

   // place value into buffer
   public void set( int value ) throws InterruptedException
   {
      System.out.printf( "Producer writes\t%2d", value );
      buffer = value;
   } // end method set

   // return value from buffer
   public int get() throws InterruptedException
   {
      System.out.printf( "Consumer reads\t%2d", buffer );
      return buffer;
   } // end method get
} // end class UnsynchronizedBuffer
```

Несинхронизирана версия на interface `Buffer`

```java
// Fig. 23.15: SharedBufferTest.java
// Application with two threads manipulating an unsynchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest
{
   public static void main( String[] args )
   {
      // create new thread pool with two threads
      ExecutorService application = Executors.newCachedThreadPool();

      // create UnsynchronizedBuffer to store ints
      Buffer sharedLocation = new UnsynchronizedBuffer();

      System.out.println(
         "Action\t\tValue\tSum of Produced\tSum of Consumed" );
      System.out.println(
         "------\t\t-----\t---------------\t---------------\n" );

      // execute the Producer and Consumer, giving each of them access
      // to sharedLocation
      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );

      application.shutdown(); // terminate application when tasks complete
   } // end main
} // end class SharedBufferTest
```

Outline

SharedBufferTest.java

(1 от 3)

Producer и Consumer споделят несихронизирания Buffer

```
Action           Value    Sum of Produced Sum of Consumed
------           -----    --------------- ---------------

Producer writes  1          1
Producer writes  2          3                               1 is lost
Producer writes  3          6                               2 is lost
Consumer reads   3                          3
Producer writes  4         10
Consumer reads   4                          7
Producer writes  5         15
Producer writes  6         21                               5 is lost
Producer writes  2         28                               6 is lost
Consumer reads   7                         14
Consumer reads   7                         21               7 read again
Producer writes  8         36
Consumer reads   8                         29
Consumer reads   8                         37               8 read again
Producer writes  9         45
Producer writes 10         55                               9 is lost

Producer done producing
Terminating Producer
Consumer reads  10                         47
Consumer reads  10                         57               10 read again
Consumer reads  10                         67               10 read again
Consumer reads  10                         77               10 read again

Consumer read values totaling 77
Terminating Consumer
```

(continued on next slide…)

Outline

SharedBufferTest .java

(2 от 3)

(continued from previous slide…)

```
Action           Value   Sum of Produced Sum of Consumed
------           -----   --------------- ---------------

Consumer reads   -1                      -1   reads -1 bad data
Producer writes   1         1
Consumer reads    1                       0
Consumer reads    1                       1   1 read again
Consumer reads    1                       2   1 read again
Consumer reads    1                       3   1 read again
Consumer reads    1                       4   1 read again
Producer writes   2         3
Consumer reads    2                       6
Producer writes   3         6
Consumer reads    3                       9
Producer writes   4        10
Consumer reads    4                      13
Producer writes   5        15
Producer writes   6        21                 5 is lost
Consumer reads    6                      19

Consumer read values totaling 19
Terminating Consumer
Producer writes   7        28                 7 never read
Producer writes   8        36                 8 never read
Producer writes   9        45                 9 never read
Producer writes  10        55                 10 never read

Producer done producing
Terminating Producer
```

Outline

SharedBufferTest.java

(3 от 3)

23.7 Задачата Производител- Потребител без синхронизация : `ArrayBlockingQueue`

- `ArrayBlockingQueue` (package `java.util.concurrent`)
  - Thread-safe структура, удобна за реализиране на общ буфер (фиксира максимален брой елементи в опашката)
  - Имплементира interface `BlockingQueue`, който е производен на interface `Queue` и има методи `put` и `take`

23.7 Задачата Производител- Потребител без синхронизация : `ArrayBlockingQueue`

- **Приложение**
  - Методът `put` добавя елемент в края на опашката `BlockingQueue`, като чака при пълна опашка
  - Методът `take` изтрива елемент от началото на `BlockingQueue`, като чака при празна опашка
  - Съхранява споделени данни в масив
  - Размерът на масива се задава в конструктора на `ArrayBlockingQueue`
  - Масивът е с фиксиран брой

Outline

```java
// Fig. 23.16: BlockingBuffer.java
// Creates a synchronized buffer using an ArrayBlockingQueue.
import java.util.concurrent.ArrayBlockingQueue;

public class BlockingBuffer implements Buffer
{
   private final ArrayBlockingQueue<Integer> buffer; //

   public BlockingBuffer()
   {
      buffer = new ArrayBlockingQueue<Integer>( 1 );
   } // end BlockingBuffer constructor

   // place value into buffer
   public void set( int value ) throws InterruptedException
   {
      buffer.put( value ); // place value in buffer
      System.out.printf( "%s%2d\t%s%d\n", "Producer writes ", value,
         "Buffer cells occupied: ", buffer.size() );
   } // end method set
```

Синхронизирана имплементация на interface `Buffer` използваща `ArrayBlockingQueue` за реализиране на синхронизация

Създава буфер с един елемент от тип `Integer` в `ArrayBlockingQueue`

`BlockingBuffer.java`

(1 от 2)

Outline

```java

   // return value from buffer
   public int get() throws InterruptedException
   {
      int readValue = 0; // initialize value read from buffer

      readValue = buffer.take(); // remove value from buffer
      System.out.printf( "%s %2d\t%s%d\n", "Consumer reads ",
         readValue, "Buffer cells occupied: ", buffer.size() );

      return readValue;
   } // end method get
} // end class BlockingBuffer
```

BlockingBuffer .java

(2 от 2)

```
// Fig. 23.17: BlockingBufferTest.java
// Two threads manipulating a blocking buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class BlockingBufferTest
{
   public static void main( String[] args )
   {
      // create new thread pool with two threads
      ExecutorService application = Executors.newCachedThreadPool();

      // create BlockingBuffer to store ints
      Buffer sharedLocation = new BlockingBuffer();

      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );

      application.shutdown();
   } // end main
} // end class BlockingBufferTest
```

Producer и Consumer споделят един и същ Buffer

Outline

BlockingBufferTest.java

(1 от 2)

Outline

```
Producer writes  1      Buffer cells occupied: 1
Consumer reads   1      Buffer cells occupied: 0
Producer writes  2      Buffer cells occupied: 1
Consumer reads   2      Buffer cells occupied: 0
Producer writes  3      Buffer cells occupied: 1
Consumer reads   3      Buffer cells occupied: 0
Producer writes  4      Buffer cells occupied: 1
Consumer reads   4      Buffer cells occupied: 0
Producer writes  5      Buffer cells occupied: 1
Consumer reads   5      Buffer cells occupied: 0
Producer writes  6      Buffer cells occupied: 1
Consumer reads   6      Buffer cells occupied: 0
Producer writes  7      Buffer cells occupied: 1
Consumer reads   7      Buffer cells occupied: 0
Producer writes  8      Buffer cells occupied: 1
Consumer reads   8      Buffer cells occupied: 0
Producer writes  9      Buffer cells occupied: 1
Consumer reads   9      Buffer cells occupied: 0
Producer writes 10      Buffer cells occupied: 1

Producer done producing
Terminating Producer
Consumer reads  10      Buffer cells occupied: 0

Consumer read values totaling 55
Terminating Consumer
```

BlockingBuffer Test.java

(2 от 2)

23.8 Producer/Consumer задача със синхронизация

- Реализация на буфера посредством `synchronized` и методите `wait`, `notify` и `notifyAllObject` на `Object`
  - Може да се използват за реализиране на условия, при което нишките да изчакват преди да продължат изпълнение
- `Нишка, която не може да продължи извиква` метода `wait на клас Object`
  - Освобождава ключа на обекта
  - Нишката преминава в Waiting и руга нишка може да получи достъп до обекта
- Нишка, която осигури изпълнението на условие по което друга нишка чака, може да информира тази нишка за изпълненото условие с извикване на метода `notify` на `Object`
  - Позволява на чакаща нишка да премине в състояние *runnable*
  - Събудената нишка може да вземе отново ключа на споделената данна
- При извикване на `notifyAll`, всички чакащи нишки преминават в състояние *runnable*

Грешка при програмиране 23.1

Грешка е да се извика `wait`, `notify` или `notifyAll` без да е придобит ключ към обект. Това води до `IllegalMonitorStateException`.

Съвет за избягване на грешки 23.1

При наличие на повече от една чакащи нишки е препоръчително да с еизползва `notifyAll` за събуждане на *waiting* нишки и преминаването им в *runnable състояние*. Това гарантира, че няма да остане неизпълнявана нишка.

```java
// Fig. 23.18: SynchronizedBuffer.java
// Synchronizing access to shared data using Object
// methods wait and notify.
public class SynchronizedBuffer implements Buffer
{
   private int buffer = -1; // shared by producer and consumer threads
   private boolean occupied = false; // whether the buffer is occupied

   // place value into buffer
   public synchronized void set( int value )
   {
      // while there are no empty locations, place thread in waiting state
      while ( occupied )
      {
         // output thread information and buffer information, then wait
         System.out.println( "Producer tries to write." );
         displayState( "Buffer full. Producer waits." );
         wait();
      } // end while

      buffer = value; // set new buffer value

      // indicate producer cannot store another value
      // until consumer retrieves current buffer value
      occupied = true;

      displayState( "Producer writes " + buffer );

      notifyAll(); // tell waiting thread(s) to enter runnable state
   } // end method set; releases lock on SynchronizedBuffer
```

Outline

SynchronizedBuffer.java

Синхронизирана версия на буфера

Синхронизиран метод, позволяващ на `Producer` да пише ако буферът е пълен

`Producer` не може да пише- изчаква

Сигнализира на `Consumer` че има нова стойност за четене

```java

   // return value from buffer
   public synchronized int get()
   {
      // while no data to read, place thr
      while ( !occupied )
      {
         // output thread information and buffer information, then wait
         System.out.println( "Consumer tries to read." );
         displayState( "Buffer empty. Consumer waits." );
         wait();
      } // end while

```

Синхронизиран метод на `Consumer` за четене на стойност, ако буферът не празен

`Consumer` не може да чете

Outline

SynchronizedBuffer.java

(2 от 3)

```
      // indicate that producer can store another value
      // because consumer just retrieved buffer value
      occupied = false;

      displayState( "Consumer reads " + buffer );

      notifyAll(); // tell waiting thread(s) to enter runnable state

      return buffer;
   } // end method get; releases lock on SynchronizedBuffer

   // display current operation and buffer state
   public void displayState( String operation )
   {
      System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
         occupied );
   } // end method displayState
} // end class SynchronizedBuffer
```

Сигнализира на `Producer` че буферът е празен

Outline

`SynchronizedBuffer.java`

(3 от 3)

Съвет за избягване на грешки 23.2

Винаги изпълнявайте `wait` в цикъл проверяващ условието, за което се чака. Възможно е нишката отново да влезе в *runnable* състояние (timed wait или `notifyAll`) преди условието да е изпълнено. Тестването на състоянието няма да позволи на нишката да се събуди, ако й е сигнализирано по- рано.

Съвет за избягване на грешки 23.2

The `wait` method "*The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the `notify` method or the `notifyAll` method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.*". *(SUN)*

Outline

SharedBuffer Test2.java

(1 от 3)

```
// Fig. 23.19: SharedBufferTest2.java
// Two threads manipulating a synchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest2
{
   public static void main( String[] args )
   {
      // create a newCachedThreadPool
      ExecutorService application = Executors.newCachedThreadPool();

      // create SynchronizedBuffer to store ints
      Buffer sharedLocation = new SynchronizedBuffer();

      System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
         "Buffer", "Occupied", "---------", "------\t\t--------" );

      // execute the Producer and Consumer tasks
      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );

      application.shutdown();
   } // end main
} // end class SharedBufferTest2
```

`Producer` и `Consumer` споделят синхронизиран буфер `Buffer`

```
Operation                                Buffer          Occupied
---------                                ------          --------

Consumer tries to read.
Buffer empty. Consumer waits.           -1              false

Producer writes 1                        1               true

Consumer reads 1                         1               false

Consumer tries to read.
Buffer empty. Consumer waits.            1               false

Producer writes 2                        2               true

Consumer reads 2                         2               false

Producer writes 3                        3               true

Consumer reads 3                         3               false

Producer writes 4                        4               true

Producer tries to write.
Buffer empty. Consumer waits.            4               true

Consumer reads 4                         4               false

Producer writes 5                        5               true

Consumer reads 5                         5               false

Producer writes 6                        6               true

Producer tries to write.
Buffer empty. Consumer waits.            6               true
```

(continued on next slide…)

<u>Outline</u>

SharedBuffer Test2.java

(2 от 3)

(continued from previous slide…)

```
Consumer reads 6                        6                false

Producer writes 7                       7                true

Producer tries to write.
Buffer full. Producer waits.            7                true

Consumer reads 7                        7                false

Producer writes 8                       8                true

Consumer reads 8                        8                false

Consumer tries to read.
Buffer empty. Consumer waits.           8                false

Producer writes 9                       9                true

Consumer reads 9                        9                false

Consumer tries to read.
Buffer empty. Consumer waits.           9                false

Producer writes 10                      10               true

Consumer reads 10                       10               false


Producer done producing
Terminating Producer

Consumer read values totaling 55
Terminating Consumer
```

Outline

**SharedBuffer
Test2.java**

(3 от 3)

23.9 Producer/Consumer Relationship: Bounded Buffers

- Cannot make assumptions about the relative speeds of concurrent
- Bounded buffer
  - Used to minimize the amount of waiting time for threads that share resources and operate at the same average speeds
  - Key is to provide the buffer with enough locations to handle the anticipated "extra" production
  - `ArrayBlockingQueue` is a bounded buffer that handles all of the synchronization details for you

Performance Tip 23.3

Even when using a bounded buffer, it is possible that a producer thread could fill the buffer, which would force the producer to wait until a consumer consumed a value to free an element in the buffer. Similarly, if the buffer is empty at any given time, a consumer thread must wait until the producer produces another value. The key to using a bounded buffer is to optimize the buffer size to minimize the amount of thread wait time, while not wasting space..

```java
// Fig. 23.20: CircularBuffer.java
// Synchronizing access to a shared three-element bounded buffer.
public class CircularBuffer implements Buffer
{
   private final int[] buffer = { -1, -1, -1 }; // shared buffer

   private int occupiedCells = 0; // count number of buffers used
   private int writeIndex = 0; // index of next element to write to
   private int readIndex = 0; // index of next element to read

   // place value into buffer
   public synchronized void set( int value ) throws InterruptedException
   {
      // output thread information and buffer information, then wait;
      // while no empty locations, place thread in bloc
      while ( occupiedCells == buffer.length )
      {
         System.out.printf( "Buffer is full. Producer waits.\n" );
         wait(); // wait until a buffer cell is free
      } // end while

      buffer[ writeIndex ] = value; // set new buffer value

      // update circular write index
      writeIndex = ( writeIndex + 1 ) % buffer.length;

      ++occupiedCells; // one more buffer cell is full
      displayState( "Producer writes " + value );
      notifyAll(); // notify threads waiting to read from buffer
   } // end method set
```

Determine whether buffer is full

Specify next write position in buffer

CircularBuffer.java

(1 of 3)

Outline

CircularBuffer .java

(2 of 3)

```java
   // return value from buffer
   public synchronized int get() throws InterruptedException
   {
      // wait until buffer has data, then read value;
      // while no data to read, place threa
      while ( occupiedCells == 0 )
      {
         System.out.printf( "Buffer is empty. Consumer waits.\n" );
         wait(); // wait until a buffer cell is filled
      } // end while

      int readValue = buffer[ readIndex ]; // read value from buffer

      // update circular read index
      readIndex = ( readIndex + 1 ) % buffer.length;

      --occupiedCells; // one fewer buffer cells are occupied
      displayState( "Consumer reads " + readValue );
      notifyAll(); // notify threads waiting to write to buffer

      return readValue;
   } // end method get

   // display current operation and buffer state
   public void displayState( String operation )
   {
      // output operation and number of occupied buffer cells
      System.out.printf( "%s%s%d)\n%s", operation,
         " (buffer cells occupied: ", occupiedCells, "buffer cells:  " );
```

Determine whether buffer is empty

Specify the next read location in the buffer

Outline

```
      for ( int value : buffer )
         System.out.printf( " %2d  ", value ); // output values in buffer

      System.out.print( "\n               " );

      for ( int i = 0; i < buffer.length; i++ )
         System.out.print( "---- " );

      System.out.print( "\n               " );

      for ( int i = 0; i < buffer.length; i++ )
      {
         if ( i == writeIndex && i == readIndex )
            System.out.print( " WR" ); // both write and read index
         else if ( i == writeIndex )
            System.out.print( " W   " ); // just write index
         else if ( i == readIndex )
            System.out.print( "  R  " ); // just read index
         else
            System.out.print( "     " ); // neither index
      } // end for

      System.out.println( "\n" );
   } // end method displayState
} // end class CircularBuffer
```

CircularBuffer.java

(3 of 3)

```java
// Fig. 23.21: CircularBufferTest.java
// Producer and Consumer threads manipulating a circular buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class CircularBufferTest
{
   public static void main( String[] args )
   {
      // create new thread pool with two threads
      ExecutorService application = Executors.newCachedThreadPool();

      // create CircularBuffer to store ints
      CircularBuffer sharedLocation = new CircularBuffer();

      // display the initial state of the CircularBuffer
      sharedLocation.displayState( "Initial State" );

      // execute the Producer and Consumer tasks
      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );

      application.shutdown();
   } // end main
} //end class CircularBufferTest
```


`Producer` and `Consumer` share the same synchronized circular `Buffer`

Outline

CircularBuffer Test.java

(2 of 5)

```
Initial State (buffer cells occupied: 0)
buffer cells:   -1   -1   -1
               ---- ---- ----
                WR

Producer writes 1 (buffer cells occupied: 1)
buffer cells:    1   -1   -1
               ---- ---- ----
                 R   W

Consumer reads 1 (buffer cells occupied: 0)
buffer cells:    1   -1   -1
               ---- ---- ----
                     WR

Buffer is empty. Consumer waits.
Producer writes 2 (buffer cells occupied: 1)
buffer cells:    1    2   -1
               ---- ---- ----
                      R   W

Consumer reads 2 (buffer cells occupied: 0)
buffer cells:    1    2   -1
               ---- ---- ----
                          WR


Producer writes 3 (buffer cells occupied: 1)
buffer cells:    1    2    3
               ---- ---- ----
                 W        R
```

(continued on next slide…)

(continued from previous slide…)

```
Consumer reads 3 (buffer cells occupied: 0)
buffer cells:    1    2    3
               ---- ---- ----
                WR

Producer writes 4 (buffer cells occupied: 1)
buffer cells:    4    2    3
               ---- ---- ----
                 R   W

Producer writes 5 (buffer cells occupied: 2)
buffer cells:    4    5    3
               ---- ---- ----
                 R         W

Consumer reads 4 (buffer cells occupied: 1)
buffer cells:    4    5    3
               ---- ---- ----
                      R    W

Producer writes 6 (buffer cells occupied: 2)
buffer cells:    4    5    6
               ---- ---- ----
                W     R
```

(continued on next slide…)

CircularBufferTest.java

(3 of 5)

(continued from previous slide…)

```
Producer writes 7 (buffer cells occupied: 3)
buffer cells:    7    5    6
               ---- ---- ----
                     WR

Consumer reads 5 (buffer cells occupied: 2)
buffer cells:    7    5    6
                ---- ---- ----
                     W    R

Producer writes 8 (buffer cells occupied: 3)
buffer cells:    7    8    6
               ---- ---- ----
                          WR

Consumer reads 6 (buffer cells occupied: 2)
buffer cells:    7    8    6
               ---- ---- ----
                 R        W

Consumer reads 7 (buffer cells occupied: 1)
buffer cells:    7    8    6
               ---- ---- ----
                      R   W

Producer writes 9 (buffer cells occupied: 2)
buffer cells:    7    8    9
               ---- ---- ----
                W     R
```

(continued on next slide…)

Outline

`CircularBufferTest.java`

(4 of 5)

(continued from previous slide…)

```
Consumer reads 8 (buffer cells occupied: 1)
buffer cells:    7    8    9
               ---- ---- ----
                W          R

Consumer reads 9 (buffer cells occupied: 0)
buffer cells:    7    8    9
               ---- ---- ----
                WR

Producer writes 10 (buffer cells occupied: 1)
buffer cells:   10    8    9
               ---- ---- ----
                 R   W


Producer done producing
Terminating Producer
Consumer reads 10 (buffer cells occupied: 0)
buffer cells:   10    8    9
               ---- ---- ----
                     WR


Consumer read values totaling: 55
Terminating Consumer
```


Outline

CircularBuffer Test.java

(5 of 5)

23.10 Types of locks

A block of code can acquire three types of lock :

- `none` at all
- an "`instance lock`", attached to a single object
- a "`static lock`", attached to a class

If a method is declared as `synchronized`, then it will acquire either `the instance lock` or `the static lock` when it is invoked, according to whether it is an `instance method` or a `static` method.

Acquiring the `instance lock` only blocks other threads from invoking a `synchronized` instance method ; it does *not* block other threads from invoking an *un-synchronized* method, nor does it block them from invoking a `static` synchronized method.

23.10 Types of locks

Similarly, acquiring the `static` lock only blocks other threads from invoking a `static` synchronized method ; it does *not* block other threads from invoking an *un-synchronized* method, nor does it block them from invoking a synchronized *instance* method.

The two types of lock have similar behaviour, but are completely independent of each other.

Outside of a method header, `synchronized(this)` acquires the `instance` lock. The `static` lock can be acquired outside of a method header in two ways :

`synchronized(Blah.class)`, using the class literal

`synchronized(this.getClass())`, if an object is available

23.10 Types of locks

Java Tutorial:

- ***A synchronized method acquires a monitor before it executes. For a `class` (`static`) method, the monitor associated with the `Class` object for the method's class is used. For an `instance method`, the `monitor` associated with this (the object for which the method was invoked) is used.***

23.10 Types of locks

Example 1:

There is no link between `synchronized static` methods and `synchronized instance methods`:

```
class A {
 static synchronized f() {...}
 synchronized g() {...}
}
```

Assume `A a = new A();`

and Thread 1 runs `A.f()` `while` Thread 2 runs `a.g()`

Then `f()` and `g()` cannot not be synchronized with each other and these thread can execute totally concurrently

23.10 Types of locks

Example 2:

To implement mutual exclusion between different instances of the object (*which is needed when accessing an external resource, for example*) use the following pattern

```
g() {
        synchronized(getClass())
          {
              ...
          }
}
```

23.10 Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

- **Introduced in Java SE 5**
- **Give programmers more precise control over thread synchronization, but are more complicated to use**
- **Any object can contain a reference to an object that implements the `Lock` interface (of package `java.util.concurrent.locks`)**
- **Call `Lock`'s `lock` method to acquire the lock**
  - **Once obtained by one thread, the `Lock` object will not allow another thread to obtain the `Lock` until the first thread releases the `Lock`**
- **Call `Lock`'s `unlock` method to release the lock**
- **All other threads attempting to obtain that `Lock` on a locked object are placed in the *waiting* state**

23.10 Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

- Class `ReentrantLock` (of package `java.util.concurrent.locks`) is a basic implementation of the `Lock` interface.
- `ReentrantLock` constructor takes a `boolean` argument that specifies whether the lock has a fairness policy
  - If `true`, the `ReentrantLock`'s fairness policy is "the longest-waiting thread will acquire the lock when it is available"—prevents starvation
  - If `false`, there is no guarantee as to which waiting thread will acquire the lock when it is available
- A thread that owns a `Lock` and determines that it cannot continue with its task until some condition is satisfied can wait on a condition object
- `Lock` objects allow you to explicitly declare the condition objects on which a thread may need to wait

23.10 Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

Note that `Lock` instances are just normal objects and can themselves be used as the target in a `synchronized` statement. Acquiring the `monitor` lock of a `Lock` instance has no specified relationship with invoking any of the `lock`() methods of that instance.

It is recommended that to avoid confusion you never use `Lock` instances in this way, except within their own implementation

23.10 Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

The `ReentrantLock.lock() - ReetrantLock.unlock()` and the `synchronized()` locking are implementation/performance wise different:

- **the `synchronized` mechanism uses a locking mechanism that is "built into" the `JVM`; the underlying mechanism is subject to the particular `JVM` implementation**
- **the `lock` classes such as `ReentrantLock` are basically coded in pure Java (via a library introduced in Java 5 which exposes CAS instructions and thread descheduling to Java) and so is somewhat more standardised across OS's and more controllable).**

23.10 Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

Functionality-wise:

- **the `synchronized` mechanism provides minimal functionality (you can `lock` and `unlock`, locking is an all-or-nothing operation, you're more subject to the algorithm the OS writers decided on), though with the advantage of built-in syntax and some monitoring built into the JVM;**
- **the `explicit` lock classes provide more control, notably you can specify a "*fair*" `lock`, lock with a `timeout`, override if you need to alter the lock's behaviour...**

23.10 Producer/Consumer Relationship: The Lock and Condition Interfaces

- `Condition` objects with `Lock`
  - Associated with a specific `Lock`
  - Created by calling a `Lock`'s `newCondition` method
- To wait on a `Condition` object, call the `Condition` 's `await` method
  - immediately releases the associated `Lock` and places the thread in the *waiting* state for that `Condition`
- Another thread can call `Condition` method `signal` to allow a thread in that `Condition`'s *waiting* state to return to the *runnable* state
  - Default implementation of `Condition` signals the longest-waiting thread
- `Condition` method `signalAll` transitions all the threads waiting for that condition to the *runnable* state
- When finished with a shared object, thread must call `unlock` to release the `Lock`

23.10 Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

- `Lock` and `Condition` may be preferable to using the `synchronized` keyword
  - `Lock` objects allow you to interrupt waiting threads or to specify a timeout for waiting to acquire a lock
  - `Lock` object is not constrained to be acquired and released in the same block of code
- `Condition` objects can be used to specify multiple conditions on which threads may wait
  - Possible to indicate to waiting threads that a specific condition object is now true

Software Engineering Observation 23.2

Using a `ReentrantLock` with a fairness policy avoids indefinite postponement.

Performance Tip 23.4

Using a `ReentrantLock` with a fairness policy can decrease program performance significantly.

Common Programming Error 23.2

Deadlock occurs when a waiting thread (let us call this thread1) cannot proceed because it is waiting (either directly or indirectly) for another thread (let us call this thread2) to proceed, while simultaneously thread2 cannot proceed because it is waiting (either directly or indirectly) for thread1 to proceed. The two threads are waiting for each other, so the actions that would enable each thread to continue execution can never occur.

Error-Prevention Tip 23.3

When multiple threads manipulate a shared object using locks, ensure that if one thread calls method `await` to enter the *waiting* state for a condition object, a separate thread eventually will call `Condition` method `signal` to transition the thread waiting on the condition object back to the *runnable* state. If multiple threads may be waiting on the condition object, a separate thread can call `Condition` method `signalAll` as a safeguard to ensure that all the waiting threads have another opportunity to perform their tasks. If this is not done, starvation might occur.

Common Programming Error 23.3

An `IllegalMonitorStateException` occurs if a thread issues an `await`, a `signal`, or a `signalAll` on a condition object without having acquired the lock for that condition object.

```java
// Fig. 23.22: SynchronizedBuffer.java
// Synchronizing access to a shared integer using the Lock and Condition
// interfaces
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public class SynchronizedBuffer implements Buffer
{
   // Lock to control synchronization with this buffer
   private final Lock accessLock = new ReentrantLock();

   // conditions to control reading and writing
   private final Condition canWrite = accessLock.newCondition();
   private final Condition canRead = accessLock.newCondition();

   private int buffer = -1; // shared by producer and consumer threads
   private boolean occupied = false; // whether buffer is occupied

   // place int value into buffer
   public void set( int value ) throws InterruptedException
   {
      accessLock.lock(); // lock this object

      // output thread information and buffer informa
      try
      {
```

Synchronized implementation of interface `Buffer` that uses `Lock`s and `Condition`s

`Condition` indicating when a producer can write

`Condition` indicating when a consumer can read

Manually acquire the lock to implement mutual exclusion

```java
         // while buffer is not empty, place thread in waiting state
         while ( occupied )
         {
            System.out.println( "Producer tries to write." );
            displayState( "Buffer full. Producer waits." );
            canWrite.await(); // wait until buffer is empty
         } // end while

         buffer = value; // set new buffer value

         // indicate producer cannot store another value
         // until consumer retrieves current buffer value
         occupied = true;

         displayState( "Producer writes " + buffer );

         // signal thread waiting t
         canRead.signal();
      } // end try
      finally
      {
         accessLock.unlock(); // unloc
      } // end finally
   } // end method set

```

Producer must wait until buffer is empty and release the lock

Producer signals the consumer that a value is available for reading

Release the lock so consumer can read

Outline

`SynchronizedBuffer.java`

(3 of 4)

```java
   // return value from buffer
   public int get() throws InterruptedException
   {
      int readValue = 0; // init
      accessLock.lock(); // lock

      // output thread informati                                en wait
      try
      {
         // while no data to read, place thread in waiting state
         while ( !occupied )
         {
            System.out.println( "Consumer tries to read." );
            displayState( "Buffer empty. Consumer waits." );
            canRead.await(); // wait until buffer is full
         } // end while

         // indicate that producer can store another value
         // because consumer just retrieved buffer value
         occupied = false;

         readValue = buffer; // retrieve value from buffer
         displayState( "Consumer reads " + readValue );

         // signal thread waiting f
         canWrite.signal();
      } // end try
```

Manually acquire the lock to implement mutual exclusion

Consumer must wait until buffer is full and release the lock

Consumer signals the producer that space is available for writing

```java
      finally
      {
         accessLock.unlock(); // unlock this object
      } // end finally

      return readValue;
   } // end method get

   // display current operation and buffer state
   public void displayState( String operation )
   {
      System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
         occupied );
   } // end method displayState
} // end class SynchronizedBuffer
```

Release the lock so producer can write

Outline

SynchronizedBuffer.java

(4 of 4)

Error-Prevention Tip 23.4

Place calls to `Lock` method `unlock` in a `finally` block. If an exception is thrown, `unlock` must still be called or deadlock could occur.

Common Programming Error 23.4

Forgetting to `signal` a waiting thread is a logic error. The thread will remain in the *waiting* state, which will prevent the thread from proceeding. Such waiting can lead to indefinite postponement or deadlock.

Outline

```
// Fig. 23.23: SharedBufferTest2.java
// Two threads manipulating a synchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest2
{
   public static void main( String[] args )
   {
      // create new thread pool with two threads
      ExecutorService application = Executors.newCachedThreadPool();

      // create SynchronizedBuffer to store ints
      Buffer sharedLocation = new SynchronizedBuffer();

      System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
         "Buffer", "Occupied", "---------", "------\t\t--------" );

      // execute the Producer and Consumer tasks
      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );

      application.shutdown();
   } // end main
} // end class SharedBufferTest2
```

```
Operation                               Buffer          Occupied
---------                               ------          --------

Producer writes 1                       1               true

Producer tries to write.
Buffer full. Producer waits.            1               true
```

(continued on next slide…)

SharedBuffer
Test2.java

(1 of 3)

(continued from prevoius slide…)

```
Consumer reads 1                        1                false

Producer writes 2                       2                true

Producer tries to write.
Buffer full. Producer waits.            2                true

Consumer reads 2                        2                false

Producer writes 3                       3                true

Consumer reads 3                        3                false

Producer writes 4                       4                true

Consumer reads 4                        4                false

Consumer tries to read.
Buffer empty. Consumer waits.           5                false

Producer writes 5                       5                true

Consumer reads 5                        5                false
```

(continued on next slide…)

Outline

SharedBufferTest2.java

(2 of 3)

(continued from previous slide…)

```
Consumer tries to read.
Buffer empty. Consumer waits.           5                false

Producer writes 6                       6                true

Consumer reads 6                        6                false

Producer writes 7                       7                true

Consumer reads 7                        7                false

Producer writes 8                       8                true

Consumer reads 8                        8                false

Producer writes 9                       9                true

Consumer reads 9                        9                false

Producer writes 10                      10               true


Producer done producing
Terminating Producer
Consumer reads 10                       10               false


Consumer read values totaling 55
Terminating Consumer
```

Outline

SharedBuffer Test2.java

(3 of 3)

23.11 Multithreading with GUI

- Event dispatch thread handles interactions with the application’s GUI components
  - All tasks that interact with an application’s GUI are placed in an event queue
  - Executed sequentially by the event dispatch thread
- Swing GUI components are not thread safe
  - Thread safety achieved by ensuring that Swing components are accessed from only the event dispatch thread—known as thread confinement
- Preferable to handle long-running computations in a separate thread, so the event dispatch thread can continue managing other GUI interactions
- Class `SwingWorker` (in package `javax.swing`) implements interface `Runnable`
  - Performs long-running computations in a worker thread
  - Updates Swing components from the event dispatch thread based on the computations’ results

Method	Description
`doInBackground`	Defines a long computation and is called in a worker thread.
`done`	Executes on the event dispatch thread when `doInBackground` returns.
`execute`	Schedules the `SwingWorker` object to be executed in a worker thread.
`get`	Waits for the computation to complete, then returns the result of the computation (i.e., the return value of `doInBackground`).
`publish`	Sends intermediate results from the `doInBackground` method to the process method for processing on the event dispatch thread.
`process`	Receives intermediate results from the `publish` method and processes these results on the event dispatch thread.
`setProgress`	Sets the progress property to notify any property change listeners on the event dispatch thread of progress bar updates.

Fig. 23.24 | Commonly used `SwingWorker` methods.

23.10 Multithreading with GUI

- **Swing GUI components**
  - **Not thread safe**
  - **Updates should be performed in the event-dispatching thread**
    - **Use `static` method `invokeLater` of class `SwingUtilities` and pass it a `Runnable` object**

Outline

RunnableObject .java

```java
// Fig. 23.17: RunnableObject.java
// Runnable that writes a random character to a JLabel
import java.util.Random;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;
import java.awt.Color;

public class RunnableObject implements Runnable
{
   private static Random generator = new Random(); // for random letters
   private Lock lockObject; // application lock; passe
   private Condition suspend; // used to suspend and r
   private boolean suspended = false; // true if threa
   private JLabel output; // JLabel for output

   public RunnableObject( Lock theLock, JLabel label )
   {
      lockObject = theLock; // store the Lock for the application
      suspend = lockObject.newCondition(); // create new Condition
      output = label; // store JLabel for outputting c
   } // end RunnableObject constructor

   // place random characters in GUI
   public void run()
   {
      // get name of executing thread
      final String threadName = Thread.currentThread().getName();

```

Implement the Runnable interface

Lock to implement mutual exclusion

Condition variable for suspending the threads

Boolean to control whether thread is suspended

Create the Lock and a Condition variable

Get name of current thread

Outline

```
      while ( true ) // infinite loop; will be terminated from outside
      {
         try
         {
            // sleep for up to 1 second
            Thread.sleep( generator.nextInt( 1000 ) );

            lockObject.lock(); // obtain the lock
            try
            {
               while ( suspended ) // loop until not suspended
               {
                  suspend.await(); // suspend thread execution
               } // end while
            } // end try
            finally
            {
               lockObject.unlock(); // unlock the lock
            } // end finally
         } // end try
         // if thread interrupted during wait/sleep
         catch ( InterruptedException exception )
         {
            exception.printStackTrace(); // print stack trace
         } // end catch

```

Obtain the lock to impose mutual exclusion

Wait while thread is suspended

Release the lock

RunnableObject.java

(2 of 4)

```
            // display character on corresponding JLabel
            SwingUtilities.invokeLater(
               new Runnable()
               {
                  // pick random character and display it
                  public void run()
                  {
                     // select random uppercase letter
                     char displayChar =
                        ( char ) ( generator.nextInt( 26 ) + 65 );

                     // output character in JLabel
                     output.setText( threadName + ": " + displayChar );
                  } // end method run
               } // end inner class
            ); // end call to SwingUtilities.invokeLater
         } // end while
      } // end method run

```

Call `invokeLater`

Method `invokeLater` is passed a `Runnable`

RunnableObject .java

(3 of 4)

```
   // change the suspended/running state
   public void toggle()
   {
      suspended = !suspended; // toggle boolean controlling state

      // change label color on suspend/resume
      output.setBackground( suspended ? Color.RED : Color.GREEN );

      lockObject.lock(); // obtain lock
      try
      {
         if ( !suspended ) // if thread resumed
         {
            suspend.signal(); // resume thread
         } // end if
      } // end try
      finally
      {
         lockObject.unlock(); // release lock
      } // end finally
   } // end method toggle
} // end class RunnableObject
```

RunnableObject.java

Obtain lock for the application

Resume a waiting thread

Release the lock

```
// Fig. 23.18: RandomCharacters.java
// Class RandomCharacters demonstrates the Runnable interface
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class RandomCharacters extends JFrame implements ActionListener
{
   private final static int SIZE = 3; // number of threads
   private JCheckBox checkboxes[]; // array of JCheckBoxes
   private Lock lockObject = new ReentrantLock( true ); // single lock

   // array of RunnableObjects to display random characters
   private RunnableObject[] randomCharacters =
      new RunnableObject[ SIZE ];

```

Create `Lock` for the application

```
   // set up GUI and arrays
   public RandomCharacters()
   {
      checkboxes = new JCheckBox[ SIZE ]; // allocate space for array
      setLayout( new GridLayout( SIZE, 2, 5, 5 ) ); // set layout

      // create new thread pool with SIZE threads
      ExecutorService runner = Executors.newFixedThreadPool( SIZE );

      // loop SIZE times
      for ( int count = 0; count < SIZE; count++ )
      {
         JLabel outputJLabel = new JLabel(); // create JLabel
         outputJLabel.setBackground( Color.GREEN ); // set color
         outputJLabel.setOpaque( true ); // set JLabel to be opaque
         add( outputJLabel ); // add JLabel to JFrame

         // create JCheckBox to control suspend/resume state
         checkboxes[ count ] = new JCheckBox( "Suspended" );

         // add listener which executes when JCheckBox is clicked
         checkboxes[ count ].addActionListener( this );
         add( checkboxes[ count ] ); // add JCheckBox to JFrame

```

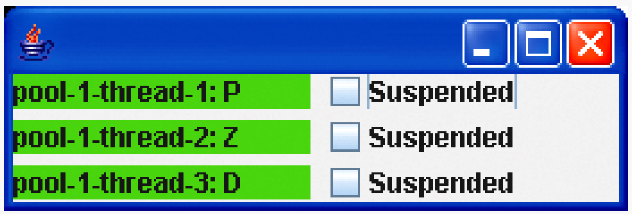
Create thread pool for executing threads

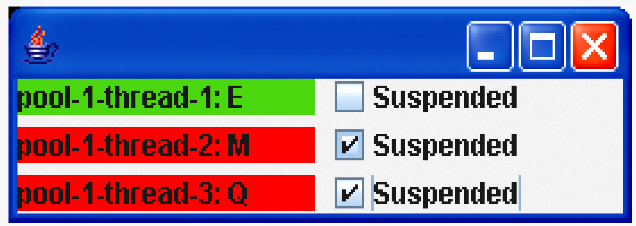
```
         // create a new RunnableObject
         randomCharacters[ count ] =
            new RunnableObject( lockObject, outputJLabel );

         // execute RunnableObject
         runner.execute( randomCharacters[ count ] );
      } // end for

      setSize( 275, 90 ); // set size of window
      setVisible( true ); // show window

      runner.shutdown(); // shutdown runner when threads finish
   } // end RandomCharacters constructor

   // handle JCheckBox events
   public void actionPerformed( ActionEvent event )
   {
      // loop over all JCheckBoxes in array
      for ( int count = 0; count < checkboxes.length; count++ )
      {
         // check if this JCheckBox was source of event
         if ( event.getSource() == checkboxes[ count ] )
            randomCharacters[ count ].toggle(); // toggle state
      } // end for
   } // end method actionPerformed

```

Execute a `Runnable`

Shutdown thread pool when threads finish their tasks

```java
    public static void main( String args[] )
    {
       // create new RandomCharacters object
       RandomCharacters application = new RandomCharacters();

       // set application to end when window is closed
       application.setDefaultCloseOperation( EXIT_ON_CLOSE );
    } // end main
} // end class RandomCharacters
```





23.11 Multithreading with GUI

- **The thread making calls to our event handlers is the event dispatch thread. This is a special thread that the GUI system sets up for performing UI tasks. Essentially, all user interface code will be executed by this special thread. Having a single designated thread handling the entire UI avoids a lot of issues that would occur if we tried to allow, say, different event handlers to be called by arbitrary threads.**
  - **if we're in our other thread and need to update the UI (e.g. to report progress to the user), we generally *need to arrange for that update code to happen in the event dispatch thread*;**
  - **by "manipulating the UI", we mean calling methods on or changing the state of any Swing components but also modifying any objects they *depend* on such as table models, cell renderers etc; firing events must also happen in the event dispatch thread.**

23.11 Multithreading with GUI

There are essentially two rules of thumb that you need to remember:

- **always manipulate your user interface from the *event dispatch thread* (with one or two safe exceptions);**
- **never block or delay the event dispatch thread- in other words, *never* call methods such as `Thread.sleep()`, `Object.wait()`, `Condition.await()` inside an event handler.**

23.11 Multithreading with GUI

Example: Supposing we have a button that launches a series of database queries. We start up a new thread so that our queries won't block the user interface:

```
JButton b = new JButton("Run query");
b.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    Thread queryThread = new Thread() {
      public void run() {
        runQueries();
      }
    };
    queryThread.start();
  }
});
```

23.11 Multithreading with GUI

But now, **from our query thread**, we want **to update a progress bar or some other component** showing the current progress to the user.

How can we do this if we're no longer in the event dispatch thread?

Answer: The `SwingUtilities` class, which provides various useful little calls, includes a method called `invokeLater()`. This method allows us to **post a "job"** to `Swing`, which it will then **run on the event dispatch thread** at its next convenience

23.11 Multithreading with GUI

```
// Called from non-UI thread
private void runQueries() {
  for (int i = 0; i < noQueries; i++) {
    runDatabaseQuery(i);
    updateProgress(i);
  }
}

private void updateProgress(final int queryNo) {
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      // Here, we can safely update the GUI
      // because we'll be called from the
      // event dispatch thread
      statusLabel.setText("Query: " + queryNo);
    }
  });
}
```

23.11 Multithreading with GUI

There's one place where it's very easy to forget that we need `SwingUtilities.invokeLater()`, and that's on application startup.

Our applications `main`() method will always be called by a special "`main`" thread that the VM starts up for us. And this main thread is *not* the event dispatch thread! So:

The code that *initialises* our GUI must also take place in an `invokeLater()`.

23.11 Multithreading with GUI

```
public class MyApplication extends JFrame {
private MyApplication() {
  // create UI here: add buttons, actions etc
}

public static void main(String[] args) {
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      MyApplication app = new MyApplication();
      app.setVisible(true);
    }
  });
}

}
```

23.11 Multithreading with GUI

- **Event dispatch thread** handles interactions with the application's GUI components
  - All tasks that interact with an application's GUI are placed in an event queue
  - Executed sequentially by the event dispatch thread
- **Swing GUI components are not thread safe**
  - Thread safety achieved by ensuring that Swing components are accessed from only the event dispatch thread—known as thread confinement
- **Preferable to handle long-running computations in a separate thread, so the event dispatch thread can continue managing other GUI interactions**
- Class `SwingWorker` (in package `javax.swing`) implements interface `Runnable`
  - Performs long-running computations in a worker thread
  - **Updates Swing components from the event dispatch thread based on the computations' results**

Method	Description
`doInBackground`	Defines a long computation and is called in a worker thread.
`done`	Executes on the event dispatch thread when `doInBackground` returns.
`execute`	Schedules the `SwingWorker` object to be executed in a worker thread.
`get`	Waits for the computation to complete, then returns the result of the computation (i.e., the return value of `doInBackground`).
`publish`	Sends intermediate results from the `doInBackground` method to the process method for processing on the event dispatch thread.
`process`	Receives intermediate results from the `publish` method and processes these results on the event dispatch thread.
`setProgress`	Sets the progress property to notify any property change listeners on the event dispatch thread of progress bar updates.

Fig. 23.24 | Commonly used `SwingWorker` methods.

23.11.1 Performing Computations in a Worker Thread

- To use a `SwingWorker`
  - Extend `SwingWorker`
  - Override methods `doInBackground` and `done`
  - `doInBackground` performs the computation and returns the result
  - `done` displays the results in the GUI after `doInBackground` returns
- `SwingWorker` is a generic class
  - First type parameter indicates the by `doInBackground` type returned
  - Second indicates the type passed between the `publish` and `process` methods to handle intermediate results
- `ExecutionException` thrown if an exception occurs during the computation

BackgroundCalculator.java

(1 of 2)

```java
// Fig. 23.25: BackgroundCalculator.java
// SwingWorker subclass for calculating Fibonacci numbers
// in a background thread.
import javax.swing.SwingWorker;
import javax.swing.JLabel;
import java.util.concurrent.ExecutionException;

public class BackgroundCalculator extends SwingWorker< String, Object >
{
   private final int n; // Fibonacci number to calculate
   private final JLabel resultJLabel; // JLabel to display the result

   // constructor
   public BackgroundCalculator( int number, JLabel label )
   {
      n = number;
      resultJLabel = label;
   } // end BackgroundCalculator constructor

   // long-running code to be run in a worker thread
   public String doInBackground()
   {
      long nthFib = fibonacci( n );
      return String.valueOf( nthFib );
   } // end method doInBackground

```

Create a subclass of SwingWorker

Possibly lengthy Fibonacci calculation to perform in the background

Outline

```
   // code to run on the event dispatch thread when doInBackground returns
   protected void done()
   {
      try
      {
         // get the result of doInBackground and
         resultJLabel.setText( get() );
      } // end try
      catch ( InterruptedException ex )
      {
         resultJLabel.setText( "Interrupted while waiting for results." );
      } // end catch
      catch ( ExecutionException ex )
      {
         resultJLabel.setText(
            "Error encountered while performing calculation." );
      } // end catch
   } // end method done

   // recursive method fibonacci; calculates nth Fibonacci number
   public long fibonacci( long number )
   {
      if ( number == 0 || number == 1 )
         return number;
      else
         return fibonacci( number - 1 ) + fibonacci( number - 2 );
   } // end method fibonacci
} // end class BackgroundCalculator
```

Display the calculation results when done

BackgroundCalculator.java

Software Engineering Observation 23.3

Any GUI components that will be manipulated by `SwingWorker` methods, such as components that will be updated from methods `process` or `done`, should be passed to the `SwingWorker` subclass's constructor and stored in the subclass object. This gives these methods access to the GUI components they will manipulate.

Outline

FibonacciNumbers.java

(1 of 5)

```
// Fig. 23.26: FibonacciNumbers.java
// Using SwingWorker to perform a long calculation with
// intermediate results displayed in a GUI.
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.border.TitledBorder;
import javax.swing.border.LineBorder;
import java.awt.Color;
import java.util.concurrent.ExecutionException;

public class FibonacciNumbers extends JFrame
{
   // components for calculating the Fibonacci of a user-entered number
   private final JPanel workerJPanel =
      new JPanel( new GridLayout( 2, 2, 5, 5 ) );
   private final JTextField numberJTextField = new JTextField();
   private final JButton goJButton = new JButton( "Go" );
   private final JLabel fibonacciJLabel = new JLabel();

   // components and variables for getting the next Fibonacci number
   private final JPanel eventThreadJPanel =
      new JPanel( new GridLayout( 2, 2, 5, 5 ) );
   private int n1 = 0; // initialize with first Fibonacci number
```

Outline

```
   private int n2 = 1; // initialize with second Fibonacci number
   private int count = 1;
   private final JLabel nJLabel = new JLabel( "Fibonacci of 1: " );
   private final JLabel nFibonacciJLabel =
      new JLabel( String.valueOf( n2 ) );
   private final JButton nextNumberJButton = new JButton( "Next Number" );

   // constructor
   public FibonacciNumbers()
   {
      super( "Fibonacci Numbers" );
      setLayout( new GridLayout( 2, 1, 10, 10 ) );

      // add GUI components to the SwingWorker panel
      workerJPanel.setBorder( new TitledBorder(
         new LineBorder( Color.BLACK ), "With SwingWorker" ) );
      workerJPanel.add( new JLabel( "Get Fibonacci of:" ) );
      workerJPanel.add( numberJTextField );
      goJButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent event )
            {
               int n;

```

FibonacciNumbers.java

(2 of 5)

Outline

```java
            try
            {
               // retrieve user's input as an integer
               n = Integer.parseInt( numberJTextField.getText() );
            } // end try
            catch( NumberFormatException ex )
            {
               // display an error message if the user did not
               // enter an integer
               fibonacciJLabel.setText( "Enter an integer." );
               return;
            } // end catch

            // indicate that the calculation has begun
            fibonacciJLabel.setText( "Calculating..." );

            // create a task to perform calculation in background
            BackgroundCalculator task =
               new BackgroundCalculator( n, fibonacciJLabel );
            task.execute(); // execute the task
         } // end method actionPerformed
      } // end anonymous inner class
   ); // end call to addActionListener
   workerJPanel.add( goJButton );
   workerJPanel.add( fibonacciJLabel );

```

FibonacciNumbers.java

(3 of 5)

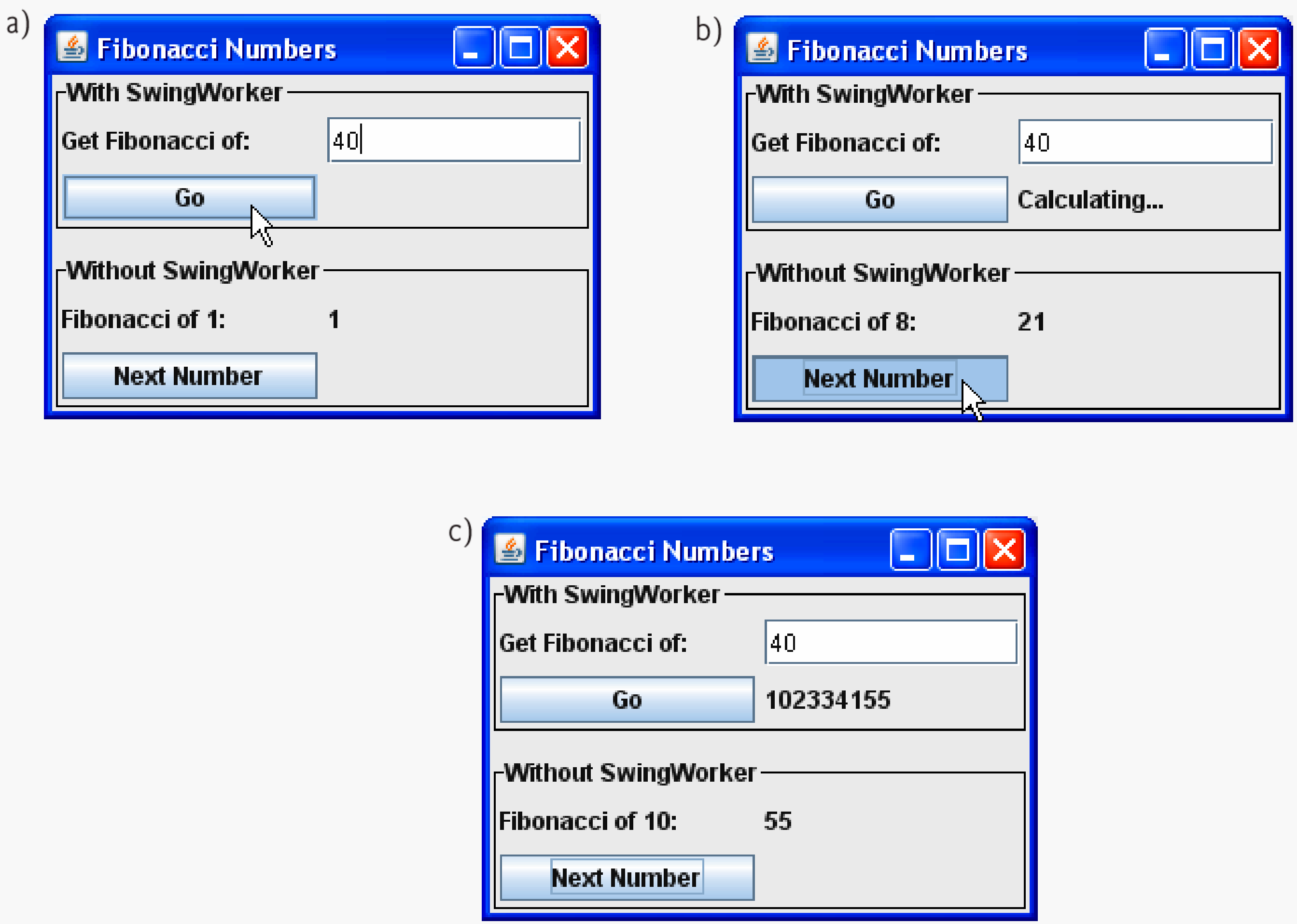
Outline

FibonacciNumbers.java

(4 of 5)

```
      // add GUI components to the event-dispatching thread panel
      eventThreadJPanel.setBorder( new TitledBorder(
         new LineBorder( Color.BLACK ), "Without SwingWorker" ) );
      eventThreadJPanel.add( nJLabel );
      eventThreadJPanel.add( nFibonacciJLabel );
      nextNumberJButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent event )
            {
               // calculate the Fibonacci number after n2
               int temp = n1 + n2;
               n1 = n2;
               n2 = temp;
               ++count;

               // display the next Fibonacci number
               nJLabel.setText( "Fibonacci of " + count + ": " );
               nFibonacciJLabel.setText( String.valueOf( n2 ) );
            } // end method actionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener
      eventThreadJPanel.add( nextNumberJButton );

      add( workerJPanel );
      add( eventThreadJPanel );
      setSize( 275, 200 );
      setVisible( true );
   } // end constructor

```

```java
    // main method begins program execution
    public static void main( String[] args )
    {
       FibonacciNumbers application = new FibonacciNumbers();
       application.setDefaultCloseOperation( EXIT_ON_CLOSE );
    } // end main
} // end class FibonacciNumbers
```



23.11.2 Processing Intermediate Results with `SwingWorker`

- `SwingWorker` methods
  - `publish` repeatedly sends intermediate results to method `process`
  - `process` executes in the event dispatch thread and receives data from method `publish` then displays the data in a GUI component
  - `setProgress` updates the progress property
- Values are passed asynchronously between `publish` in the worker thread and `process` in the event dispatch thread
- `process` is not necessarily invoked for every call to `publish`
- `PropertyChangeListener`
  - Interface from package `java.beans`
  - Defines method `propertyChange`
  - Each call to `setProgress` generates a `PropertyChangeEvent` to indicate that the progress property has changed

Outline

```java
// Fig. 23.27: PrimeCalculator.java
// Calculates the first n primes, displaying them as they are found.
import javax.swing.JTextArea;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.SwingWorker;
import java.util.Random;
import java.util.List;
import java.util.concurrent.ExecutionException;

public class PrimeCalculator extends SwingWorker< Integer, Integer >
{
   private final Random generator = new Random();
   private final JTextArea intermediateJTextArea; // displays found primes
   private final JButton getPrimesJButton;
   private final JButton cancelJButton;
   private final JLabel statusJLabel; // displays status of calculation
   private final boolean primes[]; // boolean array for finding primes
   private boolean stopped = false; // flag indicating cancelation

   // constructor
   public PrimeCalculator( int max, JTextArea intermediate, JLabel status,
      JButton getPrimes, JButton cancel )
   {
      intermediateJTextArea = intermediate;
      statusJLabel = status;
      getPrimesJButton = getPrimes;
      cancelJButton = cancel;
      primes = new boolean[ max ];

```

PrimeCalculator.java

(1 of 5)

Outline

PrimeCalculator.java

(2 of 5)

```
      // initialize all primes array values to true
      for ( int i = 0; i < max; i ++ )
         primes[ i ] = true;
   } // end constructor

   // finds all primes up to max using the Sieve of Eratosthenes
   public Integer doInBackground()
   {
      int count = 0; // the number of primes found

      // starting at the third value, cycle through the array and put
      // false as the value of any greater number that is a multiple
      for ( int i = 2; i < primes.length; i++ )
      {
         if ( stopped ) // if calculation has been canceled
            return count;
         else
         {
            setProgress( 100 * ( i + 1 ) / primes.length );

            try
            {
               Thread.currentThread().sleep( generator.nextInt( 5 ) );
            } // end try
            catch ( InterruptedException ex )
            {
               statusJLabel.setText( "Worker thread interrupted" );
               return count;
            } // end catch

```

Specify progress status as a percentage of the number of primes we are calculating

```java
            if ( primes[ i ] ) // i is prime
            {
               publish( i ); // make i available for display in prime list
               ++count;

               for ( int j = i + i; j < primes.length; j += i )
                  primes[ j ] = false; // i is not prime
            } // end if
         } // end else
      } // end for

      return count;
   } // end method doInBackground

   // displays published values in primes list
   protected void process( List< Integer > publishedVals )
   {
      for ( int i = 0; i < publishedVals.size(); i++ )
         intermediateJTextArea.append( publishedVals.get( i ) + "\n" );
   } // end method process
```

Publish each prime as it is discovered

Process all the published prime values

```java
   // code to execute when doInBackground completes
   protected void done()
   {
      getPrimesJButton.setEnabled( true ); // enable Get Primes button
      cancelJButton.setEnabled( false ); // disable Cancel button

      int numPrimes;

      try
      {
         numPrimes = get(); // retrieve doInBackground return value
      } // end try
      catch ( InterruptedException ex )
      {
         statusJLabel.setText( "Interrupted while waiting for results." );
         return;
      } // end catch
      catch ( ExecutionException ex )
      {
         statusJLabel.setText( "Error performing computation." );
         return;
      } // end catch
```

```
      statusJLabel.setText( "Found " + numPrimes + " primes." );
   } // end method done

   // sets flag to stop looking for primes
   public void stopCalculation()
   {
      stopped = true;
   } // end method stopCalculation
} // end class PrimeCalculator
```

Outline

PrimeCalculator.java

(5 of 5)

Outline

FindPrimes.java

(1 of 6)

```java
// Fig 23.28: FindPrimes.java
// Using a SwingWorker to display prime numbers and update a JProgressBar
// while the prime numbers are being calculated.
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JTextArea;
import javax.swing.JButton;
import javax.swing.JProgressBar;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.ScrollPaneConstants;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.util.concurrent.ExecutionException;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;

public class FindPrimes extends JFrame
{
   private final JTextField highestPrimeJTextField = new JTextField();
   private final JButton getPrimesJButton = new JButton( "Get Primes" );
   private final JTextArea displayPrimesJTextArea = new JTextArea();
   private final JButton cancelJButton = new JButton( "Cancel" );
   private final JProgressBar progressJProgressBar = new JProgressBar();
   private final JLabel statusJLabel = new JLabel();
   private PrimeCalculator calculator;

```

Outline

FindPrimes.java

(2 of 6)

```
   // constructor
   public FindPrimes()
   {
      super( "Finding Primes with SwingWorker" );
      setLayout( new BorderLayout() );

      // initialize panel to get a number from the user
      JPanel northJPanel = new JPanel();
      northJPanel.add( new JLabel( "Find primes less than: " ) );
      highestPrimeJTextField.setColumns( 5 );
      northJPanel.add( highestPrimeJTextField );
      getPrimesJButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent e )
            {
               progressJProgressBar.setValue( 0 ); // reset JProgressBar
               displayPrimesJTextArea.setText( "" ); // clear JTextArea
               statusJLabel.setText( "" ); // clear JLabel

               int number;

               try
               {
                  // get user input
                  number = Integer.parseInt(
                     highestPrimeJTextField.getText() );
               } // end try
```

Outline

FindPrimes.java

(3 of 6)

```
            catch ( NumberFormatException ex )
            {
               statusJLabel.setText( "Enter an integer." );
               return;
            } // end catch

            // construct a new PrimeCalculator object
            calculator = new PrimeCalculator( number,
               displayPrimesJTextArea, statusJLabel, getPrimesJButton,
               cancelJButton );

            // listen for progress bar property changes
            calculator.addPropertyChangeListener(
               new PropertyChangeListener()
               {
                  public void propertyChange( PropertyChangeEvent e )
                  {
                     // if the changed property is progress,
                     // update the progress bar
                     if ( e.getPropertyName().equals( "progress" ) )
                     {
                        int newValue = ( Integer ) e.getNewValue();
                        progressJProgressBar.setValue( newValue );
                     } // end if
                  } // end method propertyChange
               } // end anonymous inner class
            ); // end call to addPropertyChangeListener
```

```
                  // disable Get Primes button and enable Cancel button
                  getPrimesJButton.setEnabled( false );
                  cancelJButton.setEnabled( true );

                  calculator.execute(); // execute the PrimeCalculator object
               } // end method ActionPerformed
            } // end anonymous inner class
         ); // end call to addActionListener
         northJPanel.add( getPrimesJButton );

         // add a scrollable JList to display results of calculation
         displayPrimesJTextArea.setEditable( false );
         add( new JScrollPane( displayPrimesJTextArea,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER ) );

         // initialize a panel to display cancelJButton,
         // progressJProgressBar, and statusJLabel
         JPanel southJPanel = new JPanel( new GridLayout( 1, 3, 10, 10 ) );
         cancelJButton.setEnabled( false );
         cancelJButton.addActionListener(
            new ActionListener()
            {
               public void actionPerformed( ActionEvent e )
               {
                  calculator.stopCalculation(); // cancel the calculation
               } // end method ActionPerformed
            } // end anonymous inner class
         ); // end call to addActionListener
```

FindPrimes.java

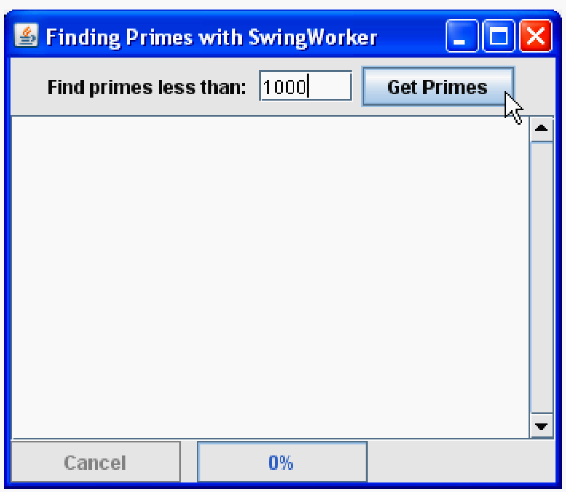
(4 of 6)

Outline

FindPrimes.java

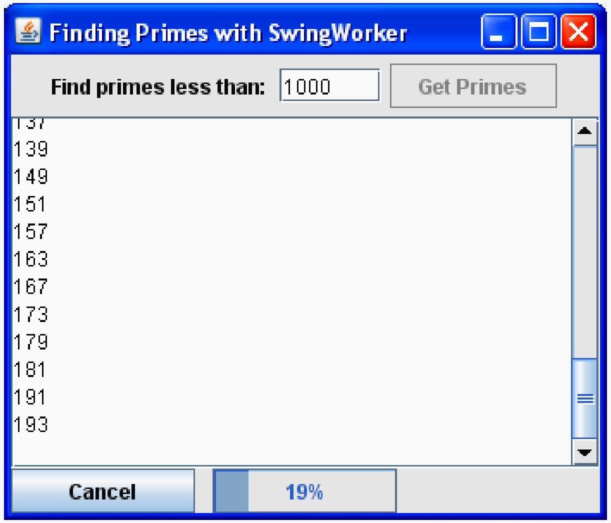

```
      southJPanel.add( cancelJButton );
      progressJProgressBar.setStringPainted( true );
      southJPanel.add( progressJProgressBar );
      southJPanel.add( statusJLabel );

      add( northJPanel, BorderLayout.NORTH );
      add( southJPanel, BorderLayout.SOUTH );
      setSize( 350, 300 );
      setVisible( true );
   } // end constructor

   // main method begins program execution
   public static void main( String[] args )
   {
      FindPrimes application = new FindPrimes();
      application.setDefaultCloseOperation( EXIT_ON_CLOSE );
   } // end main
} // end class FindPrimes
```

a)

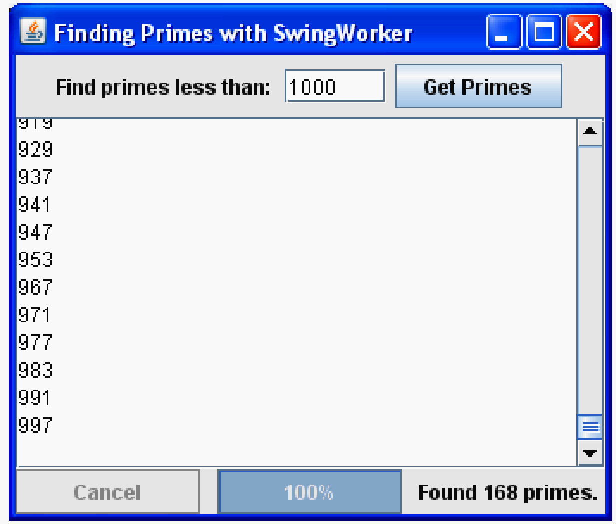


Outline

b)



c)



`FindPrimes.java`

(6 of 6)

23.12 Other Classes and Interfaces in `java.util.concurrent`

- `Callable` interface
  - package `java.util.concurrent`
  - declares a single method named `call`
  - similar to `Runnable`, but method `call` allows the thread to return a value or to throw a checked exception
- `ExecutorService` method `submit` executes a `Callable`
  - Returns an object of type `Future` (of package `java.util.concurrent`) that represents the executing `Callable`
  - `Future` declares method `get` to return the result of the `Callable` and other methods to manage a `Callable`'s execution

The Callable Interface

- **The new `java.util.concurrent.Callable` interface is much like `Runnable` but overcomes two drawbacks with `Runnable`. The `run()` method in `Runnable` cannot return a result (i.e. it returns `void`) and cannot throw a `checked` exception. If you try to throw an exception in a `run()` method, the `javac` compiler insists that you use a `throws` clause in the method signature. However, the superclass `run()` method doesn't throw an exception, so `javac` will not accept this.**

The Callable Interface

```
// External means of getting that result
public MyRunnable implements Runnable
  {
    private int fResult = 0;
    public void run () {
      ...
      fResult = 1;
    } // run
  // A getter method to provide the result
   // of the thread.

    public int getResult () { return fResult; }

} // class MyRunnable
```

The Callable Interface

The `Callable` interface solves these problems.

Instead of a `run ()` method the `Callable` interface defines a single `call()` method that takes no parameters but is allowed to throw an exception.

A simple example is

```
import java.util.concurrent.*;
public class MyCallable implements Callable<Integer>
  {
    public Integer call () throws java.io.IOException {
      return 1;
    } // Note: returns Integer!
} // MyCallable
```

The Callable Interface

Getting the return value from a `Callable` depends upon the new generics feature:

```
 FutureTask<Integer> task =
              new FutureTask<Integer> (new MyCallable ());
ExecutorService es=Executors.newFixedThreadPool (2);
es.execute (task);
try {
  int result = task.get (); // call() returns Integer!!
  System.out.println ("Result from task.get () = " +
result);
}
catch (Exception e) {
  System.err.println (e);
}
es.shutdown ();
```

Задачи

Problem 1.

Write an application that uses 100 threads to add a penny to the balance of given account. Each thread executes after sleeping for some amount of time (5ms)

1a. Write a `class Account`. It has the `balance` (assume , integer value) of an account object and the following methods:

```
int getBalance()
 // returns the current balance value,
 //which is 0 by default
void deposit(int amount)
 // makes the current thread sleep
 // for 5ms and adds  amount to balance
```

Задачи

<u>Problem 1.</u>

1b. Write a `class AccountDeposit` that creates and executes `100` threads employing a `CachedThreadPool` each of which adds a penny to a given `Account` object (make this `Account` object data member of class `AccountDeposit`. Once all the threads complete display the current balance of the given account.

Write two versions of `class AccountDeposit` - one that makes use of synchronization and another without synchronization (employing the synchronized keyword or the `Lock` interface)

Задачи

<u>Problem 2.</u>

Write an application that uses 2 cooperating threads. One of them , `DepositTask` adds a random amount in the interval `[1,10]` to a given `Account` object, while the `other` one, `WithdrawalTask` deducts a random amount in the interval `[1,10]` from the same `Account` object, when the balance of the `Account` object allows it.

Задачи

Problem 2 (*continued*).

4a. Inherit from `class Account` defined in Problem 1a. Let the newly defined class `AccountWithLock` have a reference for a Lock object and a `Condition` variable, allowing to block the `WithdrawalTask` when the balance does not allow withdrawal. Define the following methods to class `AccountWithLock`

```
public void withdraw(int amount)
 // executed by  the WithdrawalTask in
 // a separate  thread
 // output the current balance after each withdrawal
public void deposit(int amount)
 // executed by  the DepositTask in a separate  thread
 // output the current balance after each deposit
```

Задачи

<u>Problem 2</u> (*continued*) .

2b Write `class WithdrawalTask` that is `Runnable` and executes in an endless loop the `withdraw()` method of a shared `Account` object

2c Write class `DepositTask` that is `Runnable` and executes in an endless loop the `deposit()` method of a shared `Account` object

2d Write `class AcountDepositWithdraw` to create a `FixedThreadPool` for 2 threads. Execute in each of them respectively, `WithdrawalTask` and `DepositTask`