1 **Concurrency**

CST242

2 **Concurrent Processing (Page 1)**

- Only computers with *multiple processors* can truly execute multiple instructions concurrently
- On single-processor computers only a single instruction can execute at once ...
  - Older computers created the *illusion* of concurrent execution by rapidly switching between activities

3 **Concurrent Processing (Page 2)**

- Older programming languages did not enable you to specify *concurrent* activities
- Historically concurrency implemented with operating system primitives available only to experienced systems programmers
- Concurrency available in Java through the language and Java API

4 **Concurrent Processing (Page 3)**

- Single-threaded applications can lead to poor responsiveness since lengthy activities must complete before others can begin
- Multithreading can increase performance even on single-processor and multi-processor systems ...
  - When one thread cannot proceed (e.g., it is waiting for the result of an I/O operation to complete), another can use the processor (or one of the processors)

5 **Life Cycle of a Thread (Page 1)**

- Thread occupies one of several thread states
- A newly instantiated thread begins its life cycle in the new state
- When the program *starts* the thread it enters the runnable state
  - Considered to be executing its task

6 **Life Cycle of a Thread (Page 2)**

- A *runnable* thread transitions to the waiting state while it waits for another thread to perform a task ...
  - Transitions back to the *runnable* state only when another thread notifies the waiting thread to continue executing

7 **Life Cycle of a Thread (Page 3)**

- *Runnable* thread can enter timed waiting state for a specified interval of time ...
  - Transitions back to the *runnable* state when that time interval expires or when the event it is waiting for occurs

9 **Life Cycle of a Thread (Page 4)**

- *Timed waiting* and *waiting* threads cannot use a processor, even if one is available
- A *runnable* thread can transition to the *timed waiting* state if it provides an optional wait interval when it is waiting for another thread to perform a task ...
  - Returns to the *runnable* state when (1) it is notified by another thread; or (2) the

timed interval expires

10 **Life Cycle of a Thread (Page 5)**

- A thread also enters the *timed waiting* state when put to sleep
  - Remains in timed waiting state for designated period of time; then returns to *runnable* state
- A *runnable* thread transitions to blocked state when it attempts to perform a task that cannot be completed immediately …
  - Must temporarily wait until task completes
  - Cannot use a processor, even if one is available

11 **Life Cycle of a Thread (Page 6)**

- A *runnable* thread enters the terminated state (sometimes called the dead state) when it successfully completes its task …
  - Or terminates for some other reason, perhaps due to an error

12 **Java's Runnable States (Page 1)**

- At operating system level, Java's runnable state encompasses two separate states:
  - A *runnable* thread when it starts, first enters the ready state
  - When thread is dispatched by the OS it enters the running state
- Operating system hides these states from the JVM (Java virtual machine) which only sees the runnable state

13 **Java's Runnable States (Page 2)**

- Timeslicing enables the threads of equal priority to share a processor in a round-robin fashion:
  - When the thread's quantum (its timeslice) expires, returns to *ready* state
  - Operating system dispatches another thread of equal priority, if available
  - Transitions between *ready* and *running* states are handled solely by the OS

15 **Creating and Executing Threads (Page 1)**

- Objects instantiated from a class that implements Runnable interface represents a "task" that can execute *concurrently* with other tasks
  - Runnable is a member of the package java.lang (need not be imported)
- The run method (abstract method of the Runnable interface) contains code that defines task that a Runnable object performs
  - *Starting* the thread causes the object's run method to be called

16 **Creating and Executing Threads (Page 2)**

- Example:

```
public class PrintTask implements Runnable
{ …
   @Override
   public void run()
   { …
```

17 **The Thread Class**

- The Java Virtual Machine allows an application to have multiple threads of execution that are running concurrently
- The Thread class is used here to call static method sleep
- From the java.lang package (need not be imported)

18 **The sleep Method (Page 1)**

- A static method of the Thread class that causes currently executing threads to sleep:
  - Temporarily ceases execution of thread; places it into a *timed waiting* state
  - Argument specified in number of *milliseconds* (1000 milliseconds per second)

19 **The sleep Method (Page 2)**

- A static method of the Thread class that causes currently executing threads to sleep (*con*.):
  - Throws InterruptedException which is a "checked" exception (must be caught) if sleeping thread's interrupt method is called
    - Also from the java.lang package (need not be imported)

20 **The sleep Method (Page 3)**

- Format:
  Thread.sleep(*milliseconds*);
- Example:
  Thread.sleep(sleepTime);

22 **Thread Management with Executor Framework (Page 1)**

- The preferable method for managing execution of Runnable objects is to use Executor interfaces
- "Executor" objects create and manage *thread pools* (a specified number of running threads) to execute Runnables

23 **Thread Management with Executor Framework (Page 2)**

- Executor advantages over creating threads manually:
  - It can reuse existing threads to eliminate new thread overhead
  - Improves performance by optimizing number of threads to ensure that processor stays busy

24 **Thread Management with Executor Framework (Page 3)**

- The Executor method execute accepts a Runnable object as its argument ...
  - Assigns each Runnable object that it receives to one of the available threads in the thread pool
  - If none available, creates a new thread or waits for a thread to become available

25 **The ExecutorService Interface and Executor Framework (Page 1)**

- Interface ExecutorService
  - Imported from package java.util.concurrent and extends the Executor superinterface

import java.util.concurrent.ExecutorService;

- Declares methods for managing the life cycle of an Executor
- Objects of this type are created using static methods declared in class Executors (also imported from package java.util.concurrent)

  import java.util.concurrent.Executors;

26 **The ExecutorService Interface and Executor Framework (Page 2)**

- The static method newCachedThreadPool() is a "factory method" that fully implements all methods for an *ExecutorService* object
  - Including execute() and shutdown()
- Member of the class Executors which contains methods for *instantiating* objects for Executor and ExecutorService classes

27 **The ExecutorService Interface and Executor Framework (Page 3)**

- Format:

  ExecutorService *executorServiceObject* = Executors.newCachedThreadPool();
- Example:

  ExecutorService executorService = Executors.newCachedThreadPool();

28 **The execute method of the ExecutorService Interface**

- Method execute of the ExecutorService class executes command sometime *in the future*
  - Effectively starts the thread and calls the run() method when a thread becomes available
- Format:

  executorServiceObject.execute(*runnableObject*);
- Example:

  executorService.execute(task1);

29 **The shutdown method of the ExecutorService Interface**

- Method shutdown of the ExecutorService interface initiates an orderly shutdown of ExecutorService ...
  - Previously submitted tasks are completed, but no new tasks are accepted
- Format:

  executorServiceObject.shutdown();
- Example:

  executorService.shutdown();

31 **Thread Synchronization (Page 1)**

- Coordinates access to shared data by multiple concurrent threads:
  - Indeterminate results may occur unless access to a shared object is managed properly
  - Gives only one thread at a time exclusive access to code that manipulates a shared object while other threads wait
  - When thread with exclusive access to the object finishes manipulating the object, one of the threads that was waiting is allowed to proceed

32 **Thread Synchronization (Page 2)**

- Java provides built-in monitors used to implement synchronization
- Every object has a monitor and a monitor lock which enforces mutual exclusion …
  - Monitor ensures that object's monitor lock is held by a maximum of one thread at any time

33 **Thread Synchronization (Page 3)**

- To enforce mutual exclusion:
  - Thread must "acquire" the lock before it can proceed with its operation
  - Other threads attempting to perform an operation that requires the same lock will be blocked until the first thread releases the lock

34 **The synchronized Statement**

- *Within a method* enforces mutual exclusion on a block of code
- Format:

```
synchronized (object)
{
   statements
}
```

  - Where *object* is object whose monitor lock will be acquired (normally this)

35 **Synchronized Methods (Page 1)**

- A synchronized method is the equivalent of a synchronized statement that encloses the entire body of a method
- Format:

```
public synchronized void methodName( [parameters] )
{ …
```

36 **Synchronized Methods (Page 2)**

- Example:

```
public synchronized void add(int value)
{
   sharedArray[index] = value;
   index++;
}
```

37 **Synchronized Data Sharing—Making Operations Atomic**

- Simulate atomicity by ensuring that only one thread carries out a set of operations at a time
- Immutable data shared across threads
  - Declare the corresponding data fields final to indicate that variables' values will not change after they are initialized

```
private final SimpleArray sharedSimpleArray;
private final int startValue;
```

38 **The awaitTermination Method (Page 1)**

- A boolean method for an ExecutorService object that blocks until (whichever happens first):
  - Either all tasks have completed execution after a shutdown request
  - Or the timeout occurs
  - Or the current thread is interrupted
- Returns either true if the executor terminated or false if the timeout elapsed before termination

39 **The awaitTermination Method (Page 2)**

- Format:
  executorServiceObject.awaitTermination(*timeout*,*unit*)
  - *timeout*—the maximum time to wait
  - *unit*—the time unit of the *timeout* argument
- Example:
  boolean tasksEnded = executorService.awaitTermination(1, TimeUnit.MINUTES);

40 **The TimeUnit Class (Page 1)**

- The TimeUnit class represents various time level durations for concurrent operations
- Provides methods to convert *across units*, and to perform timing and delay operations in these units
- TimeUnit does not maintain time information, but only helps organize and use time representations
- Found in the java.util.concurrent package
  import java.util.concurrent.TimeUnit;

41 **The TimeUnit Class (Page 2)**

- The class provides a number of enum constants:
  TimeUnit.DAYS
  TimeUnit.HOURS
  TimeUnit.MINUTES
  TimeUnit.SECONDS
  TimeUnit.MILLISECONDS
  TimeUnit.MICROSECONDS
  TimeUnit.NANOSECONDS

43 **Software Engineering Observation 23.1**

- Place all accesses to mutable data that may be shared by multiple threads inside synchronized statements or synchronized methods that synchronize on the same lock
- When performing multiple operations on shared data, hold the lock for the entirety of the operation to ensure that the operation is effectively atomic

44 **Performance Tip 23.2**

- Keep the duration of synchronized statements as short as possible while maintaining the

needed synchronization
- Minimizes the wait time for blocked threads
- Avoid performing I/O, lengthy calculations and operations that do not require synchronization with a lock held

45 **Good Programming Practice 23.1**
- Always declare data fields that are not expected to change as final
  - Primitive variables declared as final can safely be shared across threads
  - Ensures that the object it refers to will be fully constructed and initialized before it is used by the program
  - Prevents reference from pointing to another object

46 **Multithreading with JavaFX**
- All JavaFX applications have a *single thread*, called the JavaFX application thread
  - Handles interactions with the application's controls
- All tasks requiring interaction with application's GUI are placed in an event queue
- Then tasks are executed sequentially by the JavaFX application thread as needed

47 **The Task Class (Page 1)**
- Task is an abstract class used in JavaFX to create objects that perform "long-running" computations
- *Continually updates* JavaFX components from event dispatch thread based on the computational results

48 **The Task Class (Page 2)**
- Imported from package "javafx.concurrent"
  import javafx.concurrent.Task;
- Implements interface Runnable
  - Therefore Task objects *are* threads

49 **The Task Class (Page 3)**
- To use the *generic* class Task:
  - The new class should extend the abstract class Task and …
  - *Override* Task's abstract method call

50 **The Task Class (Page 4)**
- Format:
  public class *ClassName* extends Task<*GenericType*>
  { …
- Example:
  public class FibonacciTask extends Task<Long>
  { …

51 **The Call Method (Page 1)**
- The abstract call method is invoked when Task is executed
- It must be overridden and implemented by its subclasses, and it performs the

background thread logic

- Only superclass methods updateProgress, updateMessage, and updateTitle methods of Task may be called from within method call

52 **The Call Method (Page 2)**

- Task is a *generic* class so its call method returns an object and its access modified is protected:

  @Override

  protected *GenericType* call()

  { …

  - The *GenericType* may be:
    - Long for integer types
    - Double for floating point types

53 **The Call Method (Page 3)**

- Example:

  @Override

  protected Long call()

  { …

54 **The updateMessage Method (Page 1)**

- Inherited Task method updateMessage updates Task's *message* property in the JavaFX application thread *while it is running*
- Usually placed in the call method
- *Does not wait* until the task is completed unlike method getMessage

55 **The updateMessage Method (Page 2)**

- Format:

  updateMessage(*string*);

- Example:

  updateMessage("Calculating…");

56 **The messageProperty Method (Page 1)**

- The method messageProperty in a JavaFX application thread gets the value of the message property from the task
- It is the String argument returned from the updateMessage method in superclass Task

57 **The messageProperty Method (Page 2)**

- Format:

  taskName.messageProperty()

- Example:

  labelMessage.textProperty().bind(task.messageProperty());

58 **The textProperty.bind Method (Page 1)**

- *Review*: Binding methods are used to update property values of JavaFX nodes dynamically during runtime

- The textProperty.bind method for any JavaFX control "binds" the Text property of that control to a property value in a task object
- Any time the value changes in the task, the Text property automatically updates

59 **The textProperty.bind Method (Page 2)**

- Format:
  controlName.textProperty().bind(*taskValue*);
- Example:
  labelMessage.textProperty().bind(task.messageProperty());
  - Binds the text property of labelMessage to the Task object's message

60 **The setOnRunning Method (Page 1)**

- Method setOnRunning from the Task class registers a *listener* method that is invoked when the Task "enters" the *running* state (starts to run)
  - May be registered as a *lambda* expression
- This occurs when the Task has been assigned a processor and begins executing its call method

61 **The setOnRunning Method (Page 2)**

- Format:
  taskName.setOnRunning((*eventParameter*) -> ...)
- Example:

```
task.setOnRunning( (runningEvent) ->
{
   labelFibonacci.setText("");
   buttonGo.setDisable(true);
} );
```

62 **The setOnSucceeded Method (Page 1)**

- Method setOnSucceeded from the Task class registers a *listener* method that is invoked when the Task "enters" the *succeeded* state (is completed)
  - May be registered as a *lambda* expression
- In this case, the Task's getValue method (from interface Worker) is called to obtain the result from the call method

63 **The setOnSucceeded Method (Page 2)**

- Format:
  taskName.setOnSucceeded((*eventParameter*) -> ...)
- Example:

```
task.setOnSucceeded( (succeededEvent) ->
{
   labelFibonacci.setText( task.getValue().toString() );
   buttonGo.setDisable(false);
} );
```

64 **The getValue Method**

- Method getValue from class Task returns the return value from call method when the *task is completed*
- Format:
  taskName.getValue()
- Example:

```
task.setOnSucceeded( (succeededEvent) ->
{
   labelFibonacci.setText( task.getValue().toString() )
} );
```

65 **The setCollapsible Method (Page 1)**

- For JavaFX TitledPane control, boolean method setCollapsible sets a "collapse" arrow to visible so as to collapse and hide the pane (or not)
  - Default value is true
- Set to false, lets the TitledPane be used as a "visual" container *without collapsing*
  - Only for visual effect as in "Concurrency3.java"

66 **The setCollapsible Method (Page 2)**

- Format:
  titledPaneObject.setCollapsible(true | false);
- Example:
  titledPaneWithFibonacciTask.setCollapsible(false);

68 **The updateValue Method (Page 1)**

- The inherited Task method updateValue returns a Task's *value* property in the JavaFX application thread *while it is running*
- Usually placed in call method
- *Does not wait* until the task is completed like method getValue

69 **The updateValue Method (Page 2)**

- Format:
  updateValue(*value*);
- Example:
  updateValue(i);

70 **The updateProgress Method**

- The method updateProgress returns a Task's workDone, totalWork and progress properties *while it is running*
- Format:
  updateProgress(*workDone*, *max*);
  - *workDone* is the current value from -1 to *max*
  - *max* is the terminal value
- Example:

```
updateProgress(i + 1, primes.length);
```

71 **The progressProperty Method**

- Returns a double value which represents the progress of a Task
- Format:
  taskObject.progressProperty()
- Example:
  progressBar.progressProperty().bind(task.progressProperty());

72 **The addListener Method (Page 1)**

- Method valueProperty().addListener() of class Task creates event handler that executes every time Task's *value* property updates
  - From updateValue method of the Task
- May be registered as a *lambda* expression that returns:
  - observable—value returned from the object
  - oldValue—previous value before it was updated
  - newValue—current value

73 **The addListener Method (Page 2)**

- Format:
  taskName.valueProperty().addListener((*observable*, *oldValue*, *newValue*)
  ...);
- Example:

```
task.valueProperty().addListener( (observable, oldValue, newValue) ->
{
   primes.add(newValue);
} );
```