CORE JAVA 8- DAY 5

Saratha Natarajan

AGENDA

- Today's Topic – Wrapper classes, Exceptions, Threads

WRAPPER CLASSES

- Wrapper class is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types.
- In other words, we can wrap a primitive value into a wrapper class object.

NEED OF WRAPPER CLASSES

- They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
- The classes in java.util package handles only objects and hence wrapper classes help in this case also.
- Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
- An object is needed to support synchronization in multithreading.

WRAPPER CLASSES

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

AUTOBOXING & UNBOXING

- **Autoboxing:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.
- **Unboxing:** It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

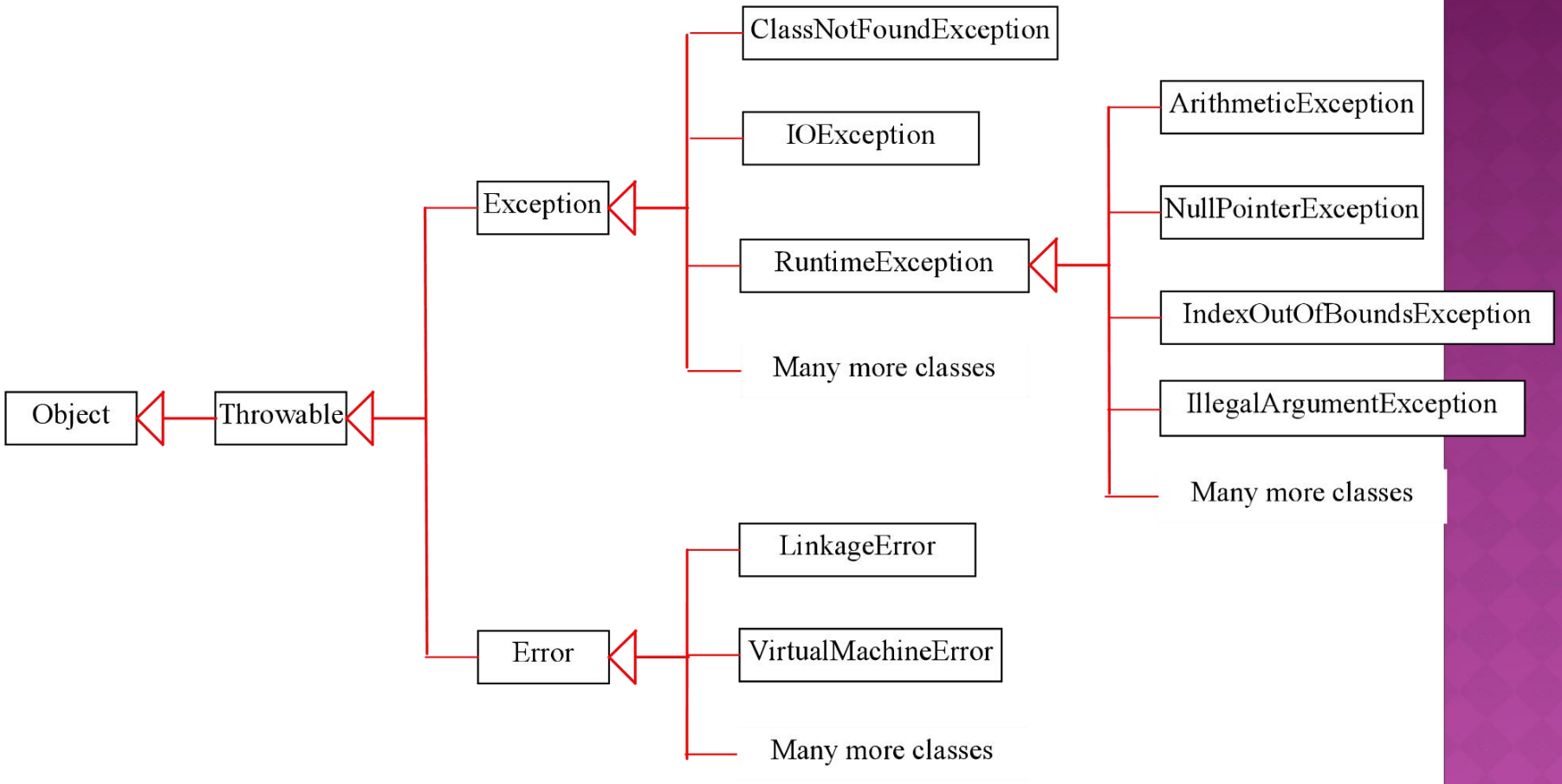
EXCEPTIONS

- Exception handling
  - Exception is an indication of problem during execution
    - “exception” occurs infrequently
    - e.g., divide by zero
  - Promotes robust and fault-tolerant software
  - Java’s Exception Handling nearly identical to C++

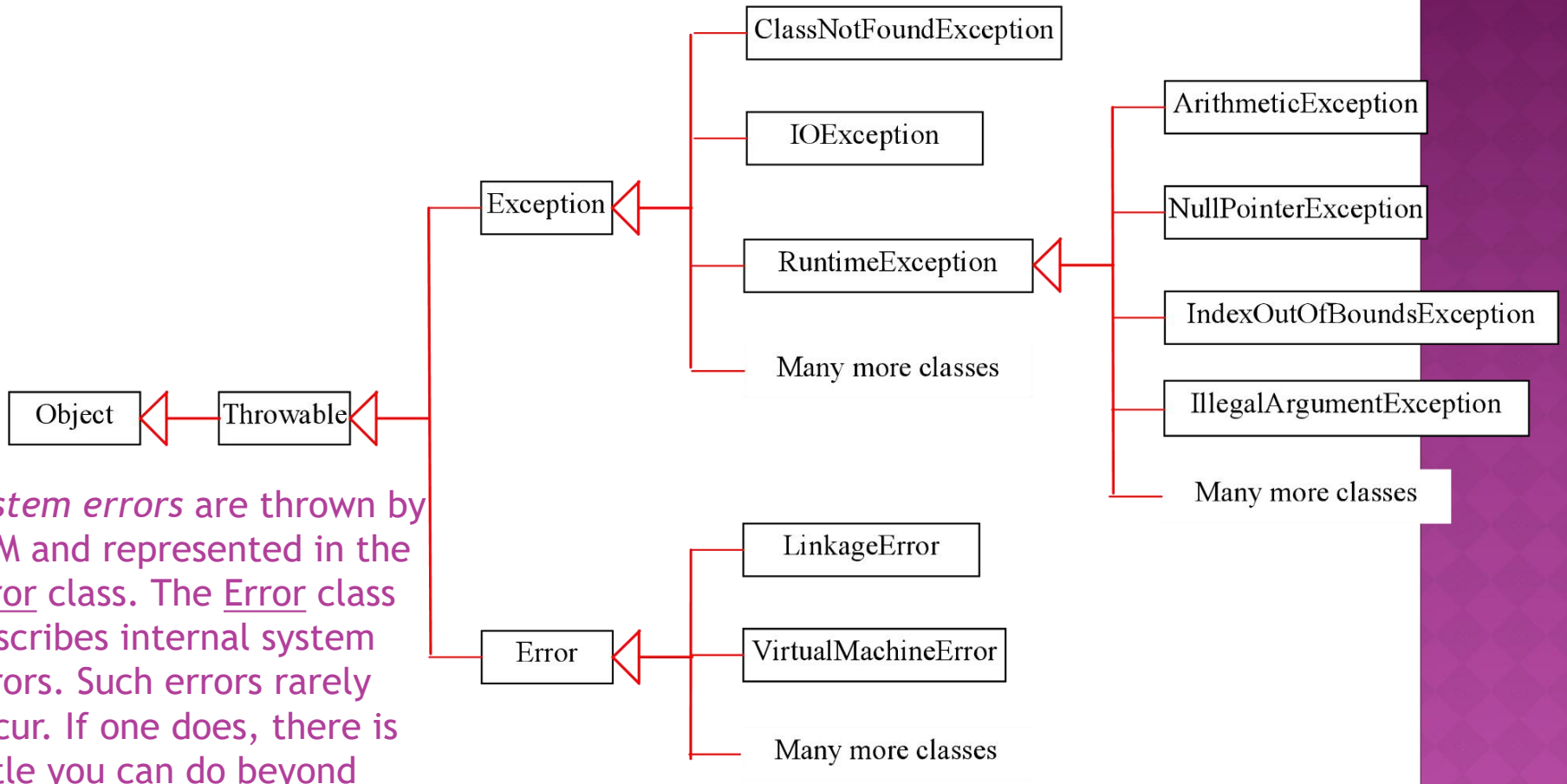
EXCEPTION-HANDLING OVERVIEW

- Mixing program code with error-handling code makes program difficult to read, modify, maintain, debug
- Uses of exception handling
  - Process exceptions from program components
  - Handle exceptions in a uniform manner in large projects
  - Remove error-handling code from “main line” of execution
- A method detects an error and throws an exception
  - Exception handler processes the error
  - Uncaught exceptions yield adverse effects
    - Might terminate program execution

EXCEPTION TYPES

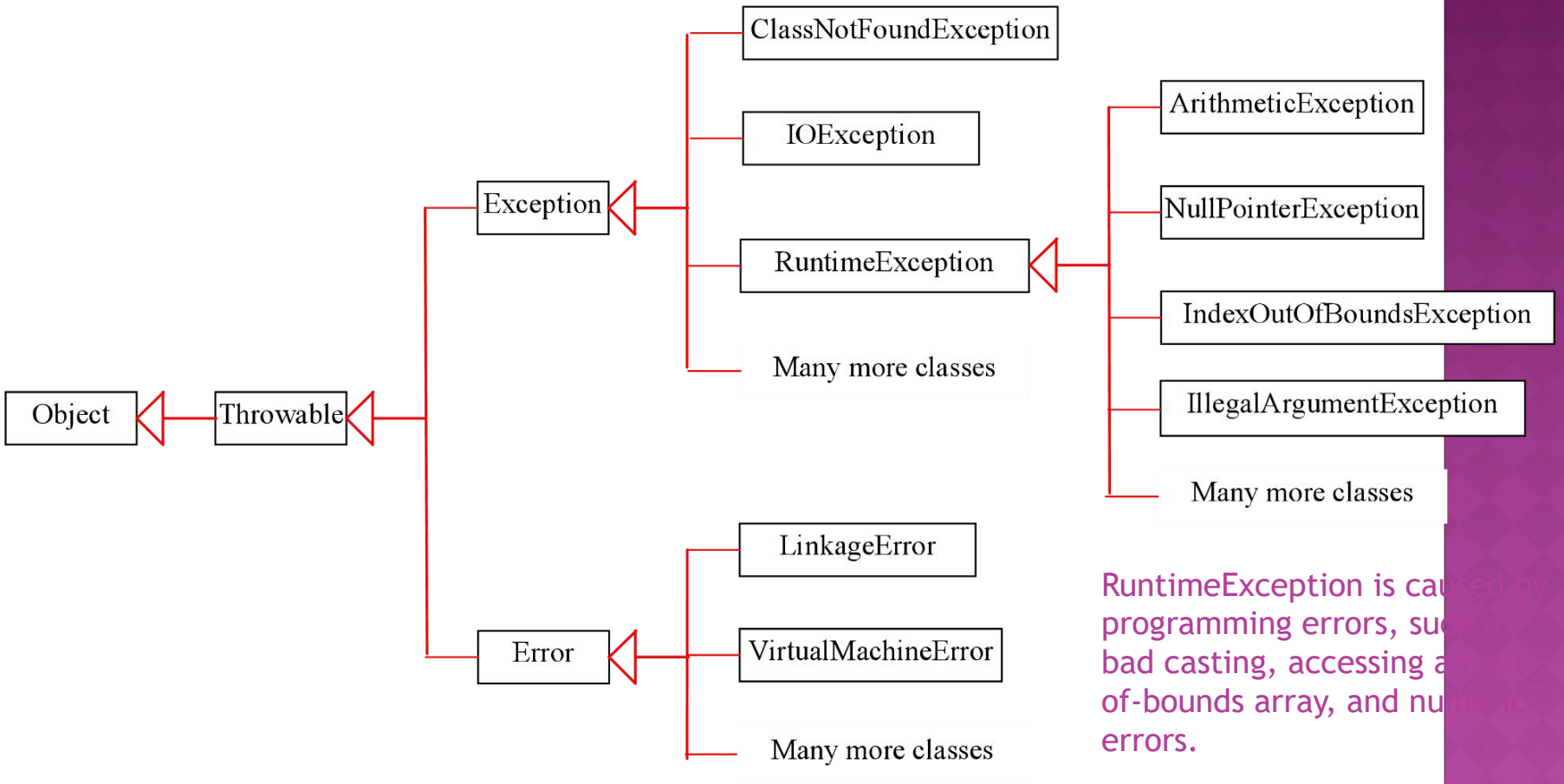


SYSTEM ERRORS



System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

RUNTIME EXCEPTIONS



RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

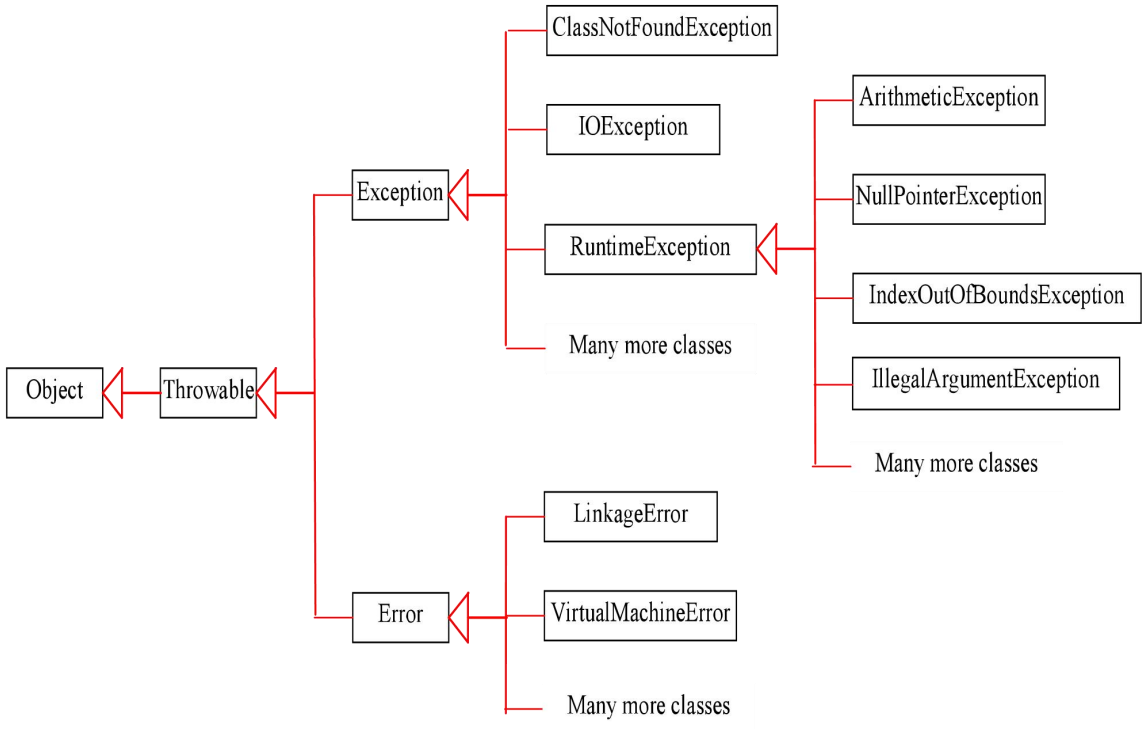
CHECKED EXCEPTIONS VS. UNCHECKED EXCEPTIONS

RuntimeException, Error and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

UNCHECKED EXCEPTIONS

In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example, a NullPointerException is thrown if you access an object through a reference variable before an object is assigned to it; an IndexOutOfBoundsException is thrown if you access an element in an array outside the bounds of the array. These are the logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in the program. To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

UNCHECKED EXCEPTIONS



TRY STATEMENT

- To process an exception when it occurs, the line that throws the exception is executed within a *try block*
- A try block is followed by one or more *catch* clauses, which contain code to process an exception
- Each catch clause has an associated exception type and is called an *exception handler*
- When an exception occurs, processing continues at the first catch clause that matches the exception type

FINALLY CLAUSE

- A try statement can have an optional clause following the catch clauses, designated by the reserved word `finally`
- The statements in the finally clause always are executed
- If no exception is generated, the statements in the finally clause are executed after the statements in the try block complete
- If an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause complete

THROW STATEMENT

- A programmer can define an exception by extending the `Exception` class or one of its descendants
- Exceptions are thrown using the *throw* statement
- Usually a throw statement is nested inside an if statement that evaluates the condition to see if the exception should be thrown

THROWS

- The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.
- Syntax of Java throws

```
modifier
return_type method_name() throws exception_class_
name{
   //method code
}
```

CHECKED EXCEPTION

- An exception is either *checked* or *unchecked*
- A *checked exception* either must be caught by a method, or must be listed in the *throws clause* of any method that may throw or propagate it
- A throws clause is appended to the method header
- The compiler will issue an error if a checked exception is not handled appropriately

UNCHECKED EXCEPTION

- An unchecked exception does not require explicit handling, though it could be processed that way
- The only unchecked exceptions in Java are objects of type `RuntimeException` or any of its descendants
- Errors are similar to `RuntimeException` and its descendants
  - Errors should not be caught
  - Errors to not require a throws clause

CUSTOM EXCEPTION

- Use the exception classes in the API whenever possible.
- Define custom exception classes if the predefined classes are not sufficient.
- Define custom exception classes by extending Exception or a subclass of Exception.

ABOUT STRINGS

- There is a special syntax for constructing strings:

  "Hello"

- Strings, unlike most other objects, have a defined *operation* (as opposed to a *method*):

  " This " + "is String " + "concatenation"

JAVA STRING CONSTRUCTOR

- | String class supports several types of constructors in Java APIs.
- The most commonly used constructors of the String class are as follows:
  - String()
  - String(String str)
  - String(char chars[ ])
  - String(char chars[ ], int startIndex, int count)
  - String(byte byteArr[ ])
  - String(byte byteArr[ ], int startIndex, int count)

- **1. String():** To create an empty String, we will call the default constructor. The general syntax to create an empty string in java program is as follows:
  - String s = new String();It will create a string object in the heap area with no value.
- **2. String(String str):** It will create a string object in the heap area and stores the given value in it. The general syntax to construct a string object with specified string str is as follows:
  - String st = new String(String str); For example: String s2 = new String("Hello Java");Here, the object s2 contains Hello Java.
- **3. String(char chars[ ]):** This constructor creates a string object and stores the array of characters in it. The general syntax to create a string object with a specified array of characters is as follows:
  - String str = new String(char char[])
  - For example:
  - char chars[ ] = { 'a', 'b', 'c', 'd' }; String s3 = new String(chars);

- **4. String(char chars[ ], int startIndex, int count):** This constructor creates and initializes a string object with a subrange of a character array.
- The argument startIndex specifies the index at which the subrange begins and count specifies the number of characters to be copied.

  String str = new String(char chars[ ], int startIndex, int count); For example: char chars[ ] = { 'w', 'i', 'n', 'd', 'o', 'w', 's' }; String str = new String(chars, 2, 3);

- **5. String(byte byteArr[ ]):** This constructor constructs a new string object by decoding the given array of bytes (i.e, by decoding ASCII values into the characters) according to the system's default character set.
- **6. String(byte byteArr[ ], int startIndex, int count):** This constructor also creates a new string object by decoding the ASCII values using the system's default character set.

USEFUL STRING METHODS I

- char charAt(int index)
  - Returns the character at the given index position (0-based)
- boolean startsWith(String prefix)
  - Tests if this String starts with the prefix String
- boolean endsWith(String suffix)
  - Tests if this String ends with the suffix String

USEFUL STRING METHODS II

- boolean equals(Object obj)
  - Tests if this String is the same as the obj (which may be any type; false if it's not a String)
- boolean equalsIgnoreCase(String other)
  - Tests if this String is equal to the other String, where case does not matter
- int length()
  - Returns the length of this string; note that this is a method, not an instance variable

USEFUL STRING METHODS III

- int indexOf(char ch)
  - Returns the position of the first occurrence of ch in this String, or -1 if it does not occur
- int indexOf(char ch, int fromIndex)
  - Returns the position of the first occurrence of ch, starting *at* (not *after*) the position fromIndex
- There are two similar methods that take a String instead of a char as their first argument

USEFUL STRING METHODS IV

- int lastIndexOf(char ch)
  - Returns the position of the last occurrence of ch in this String, or -1 if it does not occur
- int lastIndexOf(char ch, int fromIndex)
  - Returns the position of the last occurrence of ch, searching backward starting at position fromIndex
- There are two similar methods that take a String instead of a char as their first argument

USEFUL STRING METHODS V

- String substring(int beginIndex)
  - Returns a new string that is a substring of this string, beginning with the character at the specified index and extending to the end of this string.
- String substring(int beginIndex, int endIndex)
  - Returns a new string that is a substring of this string, beginning at the specified beginIndex and extending to the character at index endIndex - 1. Thus the length of the substring is endIndex-beginIndex

UNDERSTANDING "INDEX"

- With charAt(index), indexOf(x), and lastIndexOf(x), just count characters (starting from zero)

```
"She said, \"Hi\""
 0  1 2 3 4  5 6 7  8 9  10  11 12 13
```

- With substring(from) and substring(from, to), it works better to count positions *between* characters

```
"She said, \"Hi\""
 0  1 2 3 4  5 6 7  8 9 10  11 12 13  14
```

- So, for example, substring(4, 8) is "said", and substring(8, 12) is ", \"H"
- If indexOf(',') is 8, then substring(0, indexOf(',')) is "She said" and substring(indexOf(',') + 1) is " \"Hi\""

USEFUL STRING METHODS VI

- String toUpperCase()
  - Returns a new String similar to this String, in which all letters are uppercase
- String toLowerCase()
  - Returns a new String similar to this String, in which all letters are lowercase
- String trim()
  - Returns a new String similar to this String, but with whitespace removed from both ends

FINALLY, A *USELESS* STRING METHOD

- String toString()
  - Returns this String

- Why do we have this method?
  - Consistency--*Every* Object has a toString() method

STRINGS ARE IMMUTABLE

- A String, once created, cannot be changed
- *None* of the preceding methods modify the String, although several create a new String
- Statements like this create new Strings:

```
myString = myString + anotherCharacter;
```

- Creating a few extra Strings in a program is no big deal
- Creating a *lot* of Strings can be very costly

ABOUT STRINGBUFFERS

- A StringBuffer has a capacity (the number of characters it can hold) and a length (the number of characters it is currently holding)
- If the capacity is exceeded, the StringBuffer is copied to a new location with more room
- StringBuffers are used to implement String concatenation
  - Whenever you say String s = "ab" + "cd", Java creates a StringBuffer containing the characters a and b, appends the characters c and d to it, and converts the result back to a String
  - As you might guess, this isn't terribly efficient, but it's fine if you don't overdo it

STRINGBUFFER CONSTRUCTORS

- StringBuffer()
  - Constructs a StringBuffer with a capacity of 16 characters
- StringBuffer(int capacity)
  - Constructs a StringBuffer with the requested capacity
- StringBuffer(String str)
  - Constructs a StringBuffer containing the String str

USEFUL STRINGBUFFER METHODS I

- StringBuffer append(X)
  - Appends X to the end of this StringBuffer; also (as a convenience) returns this StringBuffer

- The append method is so heavily overloaded that it will work with *any* argument; if the argument is an object, its toString() method is used

USEFUL STRINGBUFFER METHODS II

- int length()
  - Returns the number of characters in this StringBuffer

- void setLength(int newLength)
  - Sets the number of characters in this StringBuffer; this may result in truncation of characters at the end, or addition of null characters

USEFUL STRINGBUFFER METHODS III

- char charAt(int index)
  - Returns the character at the location index
- void setCharAt(int index, char ch)
  - Sets the character at location index to ch
- StringBuffer reverse()
  - The sequence of characters in this StringBuffer is replaced by the reverse of this sequence, and also returned as the value of the method

USEFUL STRINGBUFFER METHODS IV

- StringBuffer insert(int offset, X)
  - Insert X starting at the location offset in this StringBuffer, and also return this StringBuffer as the value of the method. Like append, this method is heavily overloaded
- StringBuffer deleteCharAt(int index)
  - Deletes the character at location index
- StringBuffer delete(int start, int end)
  - Deletes chars at locations start through end-1

USEFUL STRINGBUFFER METHODS V

- String substring(int start)
  - Returns a new **String** of characters from this StringBuffer, beginning with the character at the specified index and extending to the end of this string.
- String substring(int start, int end)
  - Returns a new **String** of characters from this StringBuffer, beginning at location start and extending to the character at index end-1. Thus the length of the substring is end-begin
- String toString()
  - Returns the characters of this StringBuffer as a String

WHEN TO USE STRINGBUFFERS

- If you make a *lot* (thousands) of changes or additions to a String, it is much more efficient to use a StringBuffer
- If you are simply examining the contents of a String, then a String is at least as efficient as a StringBuffer
- For incidental use (such as creating output lines), use Strings; they are more convenient

ABOUT STRINGTOKENIZERS

- A StringTokenizer is used to break a string into tokens, such as words
- A StringTokenizer uses delimiters to separate tokens
  - A StringTokenizer can be made that will return the delimiters, or discard them
- You construct a StringTokenizer for a particular String, use it for *one* pass through that String, after which the StringTokenizer is "used up"
- There are only a few methods for StringTokenizers

STRINGTOKENIZER CONSTRUCTORS

- StringTokenizer(String str)
  - Constructs a tokenizer that uses the default (whitespace) delimiters " \t\n\r\f"; it does not return delimiters as tokens
- StringTokenizer(String str, String delim)
  - Constructs a tokenizer that uses the given delimiters delim; it does not return delimiters as tokens
- StringTokenizer(String str, String delim, boolean returnDelims)
  - Constructs a tokenizer that uses the given delimiters delim; it returns delimiters as tokens if returnDelims is true

STRINGTOKENIZER METHODS

- boolean hasMoreTokens()
  - Tests if this tokenizer's String has more tokens
- String nextToken()
  - Returns the next token
- String nextToken(String delim)
  - Permanently changes this tokenizer's set of delimiters, then returns the next token
- int countTokens()
  - Returns the number of tokens remaining

EXAMPLE USE OF STRINGTOKENIZER

```
StringTokenizer st =
            new StringTokenizer("this is a test");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

- Output:

```
this
is
a
test
```

AGENDA

- THREADS
- COLLECTIONS FRAMEWORK

THREADS

- To maintain responsiveness of an application during a long running task
- To enable cancellation of separable tasks
- Some problems are intrinsically parallel
- To monitor status of some resource (e.g., DB)
- Some APIs and systems demand it (e.g., Swing)

APPLICATION THREAD

- When we execute an application:
  1. The JVM creates a Thread object whose task is defined by the `main()` method
  2. The JVM starts the thread
  3. The thread executes the statements of the program one by one
  4. After executing all the statements, the method returns and the thread dies

MULTIPLE THREADS IN SAME APP

- Each thread has its private run-time stack
- If two threads execute the same method, each will have its own copy of the local variables the methods uses
- However, all threads see the same dynamic memory, i.e., heap (are there variables on the heap?)
- Two different threads can act on the same object and same static fields concurrely

CREATING THREADS

- There are two ways to create our own **Thread** object
  1. Subclassing the **Thread** class and instantiating a new object of that class
  2. Implementing the **Runnable** interface
- In both cases the **run()** method should be implemented

THREAD METHODS

void start()

- Creates a new thread and makes it runnable
- This method can be called only once

void run()

- The new thread begins its life inside this method

void stop() (deprecated)

- The thread is being terminated

THREAD METHODS

void yield()

- Causes the currently executing thread object to temporarily pause and allow other threads to execute
- Allow only threads of the same priority to run

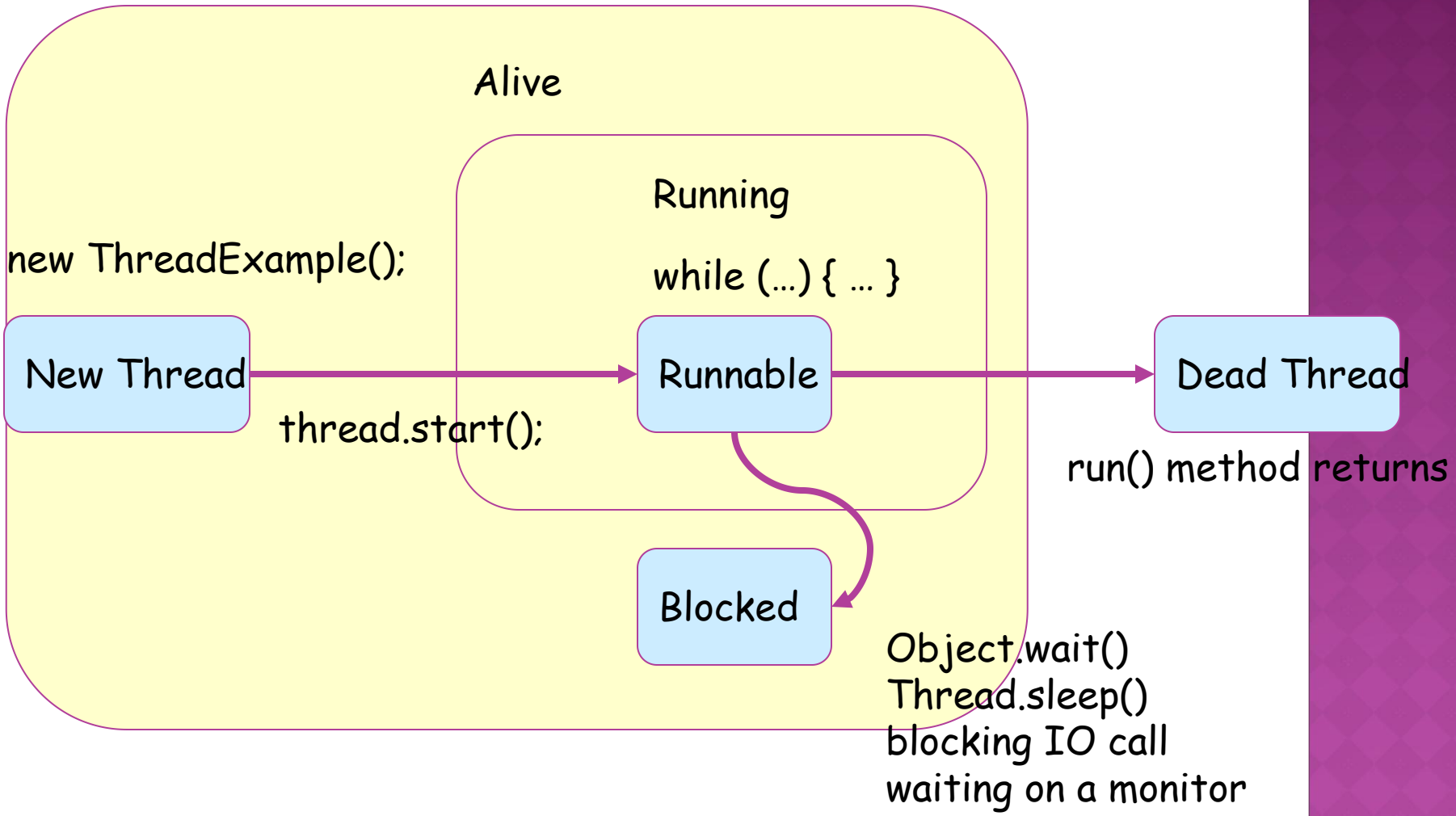
void sleep(int *m*) **or** sleep(int *m*, int *n*)

- The thread sleeps for *m* milliseconds, plus *n* nanoseconds

RUNNABLE OBJECT

- When running the Runnable object, a Thread object is created from the Runnable object
- The Thread object's **run()** method calls the Runnable object's **run()** method
- Allows threads to run inside any object, regardless of inheritance

THREAD STATE DIAGRAM



THREADGROUP

- ThreadGroup creates a group of threads. It offers a convenient way to manage groups of threads as a unit. This is particularly valuable in situation in which you want to suspend and resume a number of related threads.
- The thread group form a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group but not to access information about its thread group's parent thread group or any other thread group.

- Constructors
- **public ThreadGroup(String name):** Constructs a new thread group. The parent of this new group is the thread group of the currently running thread.**Throws:** SecurityException - if the current thread cannot create a thread in the specified thread group.
- **public ThreadGroup(ThreadGroup parent, String name):** Creates a new thread group. The parent of this new group is the specified thread group.**Throws:** NullPointerException - if the thread group argument is null. SecurityException - if the current thread cannot create a thread in the specified thread group.

THREAD PRIORITY

- Every thread has a priority
- When a thread is created, it inherits the priority of the thread that created it
- The priority values range from 1 to 10, in increasing priority

THREAD PRIORITY (CONT.)

- The priority can be adjusted subsequently using the **setPriority()** method
- The priority of a thread may be obtained using **getPriority()**
- Priority constants are defined:
  - MIN_PRIORITY=1
  - MAX_PRIORITY=10
  - NORM_PRIORITY=5

The main thread is created with priority NORM_PRIORITY

DAEMON THREADS

- Daemon threads are “background” threads, that provide services to other threads, e.g., the garbage collection thread
- The Java VM will not exit if non-Daemon threads are executing
- The Java VM will exit if only Daemon threads are executing
- Daemon threads die when the Java VM exits
- Q: Is the **main** thread a daemon thread?

SYNCHROZINATION

- Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.
- So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point in time.
- Java provides a way of creating threads and synchronizing their tasks using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword.
- A synchronized block in Java is synchronized on some object. All synchronized blocks synchronize on the same object can only have one thread executing inside them at a time.
- All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

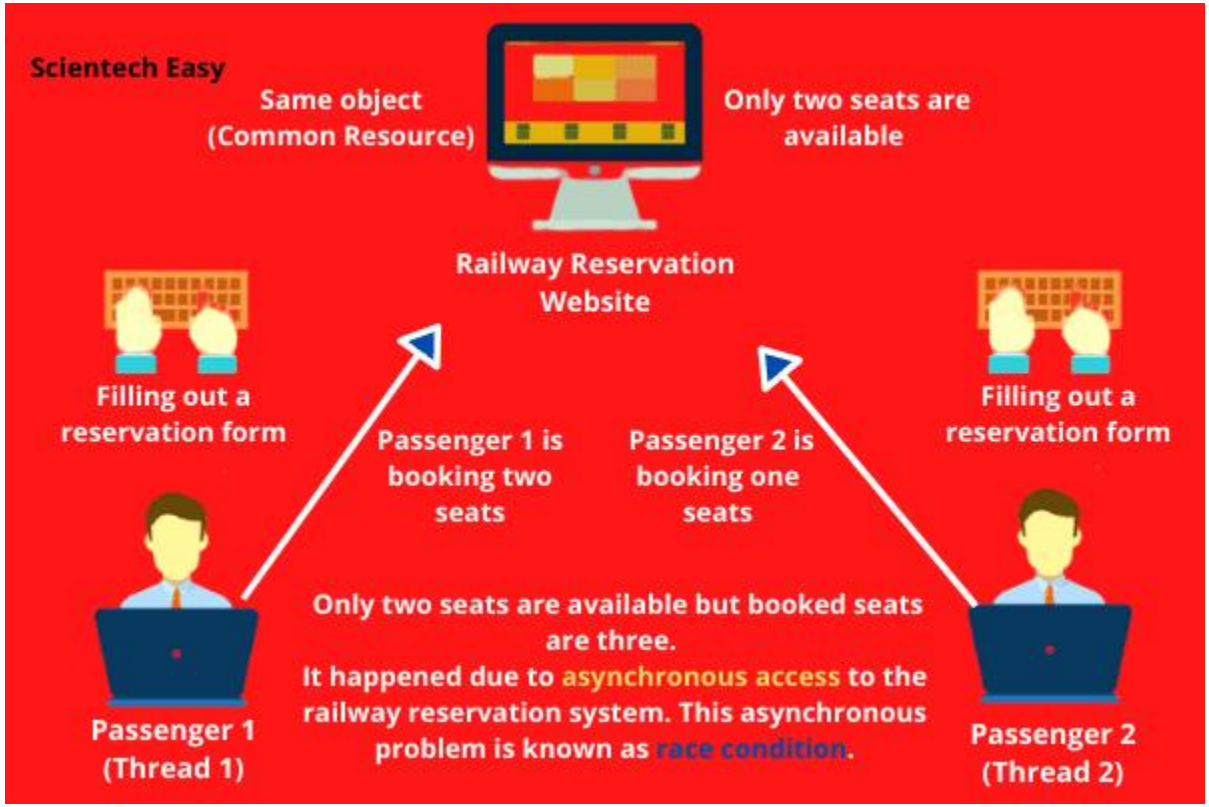
- Following is the general form of a synchronized block:
- // Only one thread can execute at a time.
- // sync_object is a reference to an object
- // whose lock associates with the monitor.
- // The code is said to be synchronized on
- // the monitor object
- synchronized(sync_object) {
- // Access shared variables and other
- // shared resources
- }
- This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

SHARED RESOURCES

- If one thread tries to read the data and other thread tries to update the same date, it leads to inconsistent state.
- This can be prevented by synchronising access to the data.
- Use “Synchronized” method:
  - public synchronized void update()
  - {
    - …
  - }

SHARED RESOURCES

- Applications Access to Shared Resources need to be coordinated.
  - Printer (two person jobs cannot be printed at the same time)
  - Simultaneous operations on your bank account.
  - Can the following operations be done at the same time on the same account?
    - Deposit()
    - Withdraw()
    - Enquire()


Scientech Easy
Same object
(Common Resource)
Only two seats are
available
Railway Reservation
Website
Filling out a
reservation form
Filling out a
reservation form
Passenger 1 is
booking two
seats
Passenger 2 is
booking one
seats
Only two seats are available but booked seats
are three.
It happened due to asynchronous access to the
railway reservation system. This asynchronous
problem is known as race condition.
Passenger 1
(Thread 1)
Passenger 2
(Thread 2)

THREAD VS RUNNABLE

Extending Thread	Implementing Runnable
1. It cannot extend other class, if you extend Thread.	1. If we implemented Runnable then your Thread class can still extend another class.
2. If you extend Thread then they are tightly coupled.	2. If you implement Runnable, both Task and Executor are loosely coupled.
3. No code reusability, other than copy and paste.	3. Gives better code reusability.
4. Overhead of additional method.	4. No overhead of any method.
5. Maintanance of code is not easy.	5. Maintanance of code is lot easier
6. Each thread creates a unique object and gets associated with it.	6. Multiple threads share the same objects.
7. As each thread create a unique object, more memory is required.	7. As multiple threads share the same object less memory is used.
8. suspend(), resume() are not available in Runnable then we prefer Thread class.	8. To achieve basic funtionality, you can simply implement Runnable interface.

DAY – 4 OVER

Activities- FLOW CONTROL , ARRAYS