String Handling

- String s = new String(); will create an instance of **String** with no characters in it.

  char chars[] = { 'a', 'b', 'c' };

  String s = new String(chars);

- Once a string object has been created, we cannot change the characters in that.
- This constructor initializes **s** with the string “abc”.

  char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

  String s = new String(chars, 2, 3);

- This initializes **s** with the characters **cde**.

```
// Construct one String from another.
class MakeString
{
public static void main(String args[])
{
        char c[] = {'J', 'a', 'v', 'a'};
        String s1 = new String(c);
        String s2 = new String(s1);
        System.out.println(s1);
        System.out.println(s2);
}
}
```

The output from this program is as follows:

```
Java
Java
```

- Even though Java's **char** type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set.
- Because 8-bit ASCII strings are common, the **String** class provides constructors that initialize a string when given a **byte** array.
- String(byte *asciiChars*[ ])
- String(byte *asciiChars*[ ], int *startIndex*, int *numChars*)
- Here, *asciiChars* specifies the array of bytes.
- *The contents of the array are copied whenever you create a **String** object from an array. If you modify the contents of the array after you have created the string, the **String** will be unchanged.*

```
// Construct string from subset of char array.
class SubStringCons
{
public static void main(String args[])
{
        byte ascii[] = {65, 66, 67, 68, 69, 70 };
        String s1 = new String(ascii);
        System.out.println(s1);
        String s2 = new String(ascii, 2, 3);
        System.out.println(s2);
}
}
```

This program generates the following output:

```
ABCDEF
CDE
```

String length

- char chars[] = { 'a', 'b', 'c' };
- String s = new String(chars);
- **System.out.println(s.length()**);

- When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String**.
- **valueOf( )** is overloaded for all the simple types and for type **Object**.
- For the simple types, **valueOf( )** returns a string that contains the human-readable equivalent of the value with which it is called.
- For objects, **valueOf( )** calls the **toString( )** method on the object.
- Override toString().
- **Box**'s **toString( )** method is automatically invoked when a **Box** object is used in a concatenation expression or in a call to **println( )**.

Character Extraction

1. **charAt( )**

   To extract a single character from a **String**

   char charAt(int *where*)

   Here, *where* is the index of the character that we want to obtain

   char ch;

   ch = "abc".charAt(1);

   assigns the value “**b**” to **ch**.

2. **getChars( )**

If we need to extract more than one character at a time, we can use the **getChars( )** method.

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

- *sourceStart* specifies the index of the beginning of the substring
- *sourceEnd* specifies an index that is one past the end of the desired substring.
- The array that will receive the characters is specified by *target.*
- The index within *target* at which the substring will be copied is passed in *targetStart.*
- *Example*

3. getBytes( )

- There is an alternative to **getChars( )** that stores the characters in an array of bytes.
- it uses the default character-to-byte conversions provided by the platform.
- byte[ ] getBytes( )
- **getBytes( )** is most useful when we are exporting a **String** value into an environment that does not support 16-bit Unicode characters.

4. toCharArray( )

- To convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**.
- It returns an array of characters for the entire string.
- char[ ] toCharArray( )

String Comparison

1. **equals( ) and equalsIgnoreCase( )**

boolean equals(Object *str*)

str is the **String** object being compared with the invoking **String** object.

The comparison is case-sensitive.

boolean equalsIgnoreCase(String *str*)

To perform a comparison that ignores case differences

Example

2. regionMatches( )

- The **regionMatches( )** method compares a specific region inside a string with another specific region in another string.

- boolean regionMatches(int *startIndex*, String *str2*,int *str2StartIndex*, int *numChars*)

- boolean regionMatches(boolean *ignoreCase*,int *startIndex*, String *str2*,int *str2StartIndex*, int *numChars*)

3. startsWith( ) and endsWith( )

- The **startsWith( )** method determines whether a given **String** begins with a specified string.
- **endsWith( )** determines whether the **String** in question ends with a specified string.
- boolean startsWith(String *str*)
- boolean endsWith(String *str*)
- "Foobar".endsWith("bar")
- and
- "Foobar".startsWith("Foo")
- are both **true**.
- boolean startsWith(String *str*, int *startIndex*)
- "Foobar".startsWith("bar", 3)

4. compareTo( )

- For sorting applications, we need to know which is *less than, equal to,* or *greater than* the next.
- The **String** method **compareTo( )** serves this purpose.
- int compareTo(String *str*)

Value	Meaning
Less than zero	The invoking string is less than *str.*
Greater than zero	The invoking string is greater than *str.*
Zero	The two strings are equal.

Example

- int compareToIgnoreCase(String *str*)

Searching Strings

- **indexOf( )** Searches for the first occurrence of a character or substring.
- **lastIndexOf( )** Searches for the last occurrence of a character or subs
- To search for the first occurrence of a character, use
  - int indexOf(int *ch*)
- To search for the last occurrence of a character, use
  - int lastIndexOf(int *ch*)
- To search for the first or last occurrence of a substring, use
  - int indexOf(String *str*)
  - int lastIndexOf(String *str*)

- int indexOf(int *ch*, int *startIndex*)
- int lastIndexOf(int *ch*, int *startIndex*)
- int indexOf(String *str*, int *startIndex*)
- int lastIndexOf(String *str*, int *startIndex*)
- Here, *startIndex* specifies the index at which point the search begins
- For **indexOf( )**, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )**, the search runs from *startIndex* to zero.
- Example

Modifying a String

- **String** objects are immutable, whenever we want to modify a **String**, we must either copy it into a **StringBuffer** or **StringBuilder**, or use one of the following **String** methods, which will construct a new copy of the string with our modifications complete.

1. **substring( )**

- We can extract a substring using **substring( )**.

    String substring(int *startIndex*)

- Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

    String substring(int *startIndex*, int *endIndex*)

- Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. Not including, the ending index.

2. concat( )

- We can concatenate two strings using **concat( )**.

  String concat(String *str*)

- This method creates a new object that contains the invoking string with the contents of *str* appended to the end.
- **concat( )** performs the same function as **+**.
- String s1 = "one";
- String s2 = s1.concat("two");

3. replace( )

String replace(char *original*, char *replacement*)

- Replaces all occurrences of one character in the invoking string with another character.
- Here, *original* specifies the character to be replaced by the character specified by *replacement*.

String s = "Hello".replace('l', 'w');

puts the string “Hewwo” into **s**.

String replace(CharSequence *original*, CharSequence *replacement*)

- Replaces one character sequence with another.

4. trim( )

- The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

  String trim( )

- Here is an example:

  String s = “ Hello World ".trim();

- This puts the string “Hello World” into **s**.

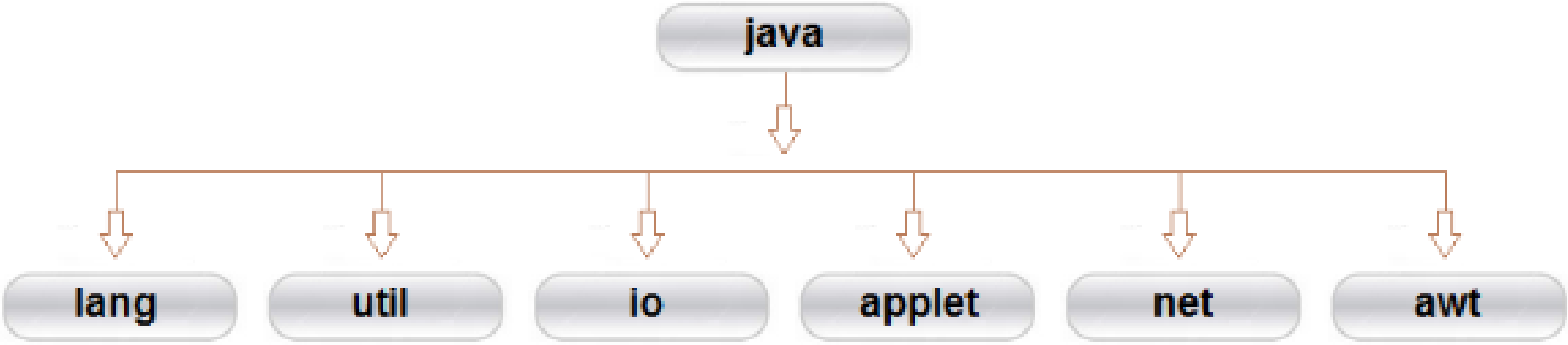
Changing the Case of Characters Within a String

- String toLowerCase( )
- String toUpperCase( )
- String s = "This is a test.";
- String upper = s.toUpperCase();
- Uppercase: THIS IS A TEST.
- String lower = s.toLowerCase();
- Lowercase: this is a test.

Packages

- *Packages* are containers for classes that are used to keep the class name space compartmentalized.
- A package allows we to create a class named **List**, which we can store in our own package without concern that it will collide with some other class named **List** stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- Java provides a mechanism for partitioning the class name space into more manageable chunks.
- The package is both a naming and a visibility control mechanism.

Java API Packages



- Java application programming interface (API) is a list of all classes that are part of the Java development kit (JDK).
- It includes all Java packages, classes, and interfaces, along with their methods, fields, and constructors.
- These prewritten classes provide a tremendous amount of functionality to a programmer.
- A programmer should be aware of these classes and should know how to use them.

java.lang	Language support classes. They include classes for primitive types, string, math functions, thread and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, data, etc.
java.io	Input/output support classes. They provide facilities for the input and output of data.
java.applet	Classes for creating and implementing applets.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

Defining Packages

- Simply include a **package** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- If we omit the **package** statement, the class names are put into the default package, which has no name.

  package *pkg*;

- Here, *pkg* is the name of the package.
- Java uses file system directories to store packages.

- package MyPackage;
- For example, the **.class** files for any classes we declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.
- Case is significant, and the directory name must match the package name exactly.
- More than one file can include the same **package** statement.
- The general form of a multileveled package statement is here:

  package *pkg1*[.*pkg2*[.*pkg3*]];

  package java.awt.image;
- needs to be stored in **java\awt\image** in a Windows environment.
- Example

Importing Packages

- Java includes the **import** statement to bring certain classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name.
- **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.

  import *pkg1*[.*pkg2*].(*classname*|*);

- Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**).
- Finally, you specify either an explicit *classname* or a star (*****), which indicates that the Java compiler should import the entire package.

- All of the standard Java classes included with Java are stored in a package called **java**.
- The basic language functions are stored in a package inside of the **java** package called **java.lang**.
- It is implicitly imported by the compiler for all programs.

Exception Handling

Exception

- An exception is an abnormal condition that arises in a code sequence at run time
- A Java exception is an object that describes an exceptional condition that occurred in a piece of code
- When an exception arises, an exception object is created and thrown in the method that caused the error
- Generated exception should caught and processed by the code
- Exception can generated by
  - Java Runtime System (Errors caused by violation of language rules or by constraints of execution environment)
  - Manually generated by user code (Used to report some error condition to the caller of a method)

Exception Handling

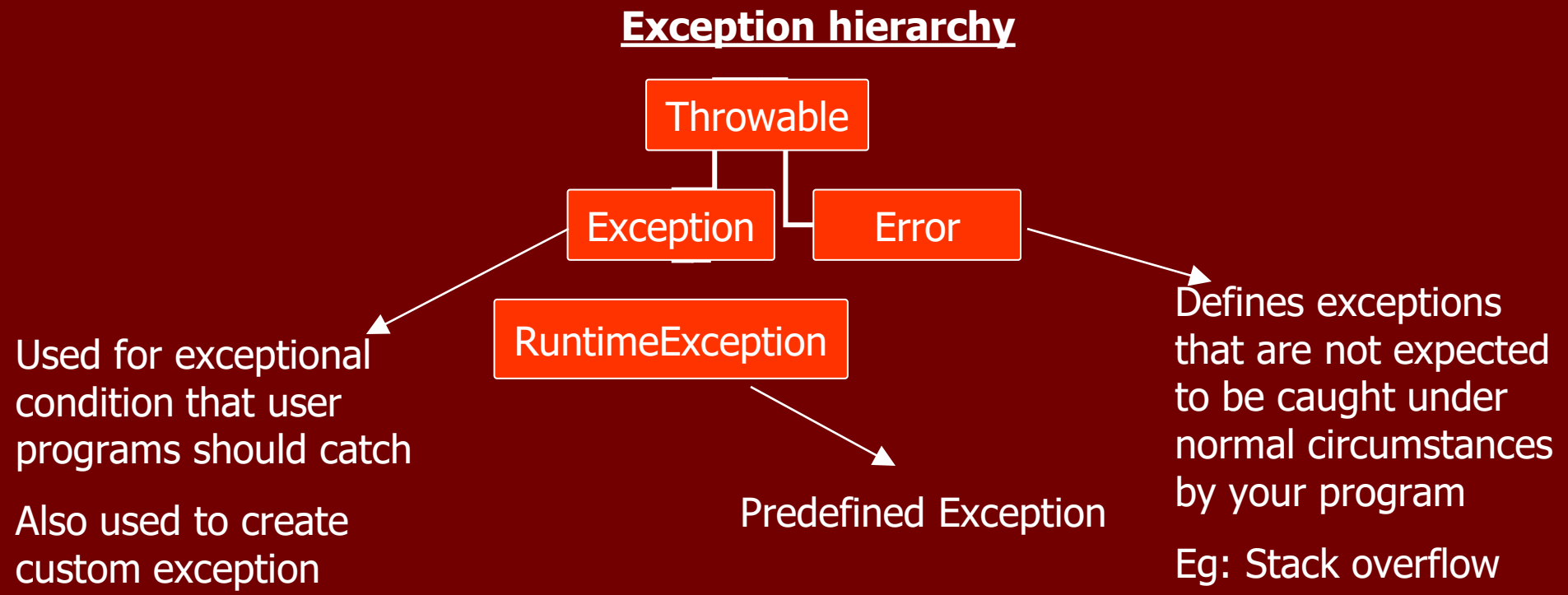
- Step 1: Program statements that can generate error will enclose within a **try** block
- Step 2: When exception occurs within try block, it will **thrown**
  - System generated exceptions are automatically thrown by Java Runtime
  - Manual exceptions should explicitly thrown by **throw** statement
  - Any exception thrown out of a method must be specified by using **throws** clause
- Step 3: Generated exception should catch and should handle by user code
- Step 4: Any code that must execute after try block is put in **finally** block

General Form

```
try
{
        //block of code to monitor for errors
}
catch(ExceptionType1  exObj)
{
        //exception handler for Exception type1
}
catch(ExceptionType2  exObj)
{
        //exception handler for Exception type2
}
// ..........
finally
{
        //block of code to be executed after try block ends
}
```

Exception Types

- All exception types are subclasses of the built-in class **Throwable**



Uncaught Exceptions

- When an exception is not handled by the user, Java run-time system constructs a new exception objects and throws
- Unhandled exception causes termination of program execution
- Uncaught exception will handle by default handler
- Default handler displays a string describing the exception, prints a stack trace from which the point the exception occured

Using try and catch

- Exception handling helps to
  - Fix the error
  - Prevents the program from automatically terminating
- When an exception occurs inside the try block, control will transfer to catch block
- After executing catch block, execution continues with rest of the statements after try..catch block
- try..catch forms a unit and all try should have atleast one catch block
- Catch statement cannot catch an exception thrown by another try statement

Exception Object

- An exception object can directly print by using println() method.
- A toString() method will automatically invoke to get exception description from exception object

Multiple catch Clauses

- In order to handle multiple exception generated from a try block, a try block can preceed more than one catch block
- Multiple catch block will contain different Exception classes
- Exception sub-classes should come before exception superclass
- When an exception is thrown,
  - Each catch block will inspect to find a matching block for the generated exception
  - Statements in the matching catch block will execute
  - Bypass rest of the catch blocks
  - Execute statements in finally block and goes to the immediate statement in the sequence

Nested try statements

- A try statement inside the block of another try statement
- If an inner try block does not have a matching catch block for a particular exception
  - The next try statement's catch block will inspected for a match
  - Above process continues until a match is found or exited from nested try block
  - If no matching catch found, default exception handler will take over the duty

Throw statement

- Manually generated exceptions are thrown by using **throw** statement
- General Format
  - throw ThrowableInstance;
  - ThrowableInstance is an instance of Throwable class or subclass of Throwable
- Primitive types and non-Throwable classes(String and objects) cannot use as exception
- Throwable object are created from parameter in catch block or by using **new** operator
- Flow of execution stops after throw statement, nearest try block is inspected to see matching catch block

Throws statement

- A method capable of causing an exception can defined using throws clause in the method's declaration
- It helps the caller of the method to guard themself against the exception
- A throws clause lists the types of exceptions that a method might throw
- Error, RuntimeException or any of their subclasses are not used with throws
- All other exceptions that a method can throw must be declared in the throws clause

General Format

Type method_name(parameter-list) throws exception-list

{

//body of method

}

- exception-list is a comma separated list of the exceptions that a method can throw

finally

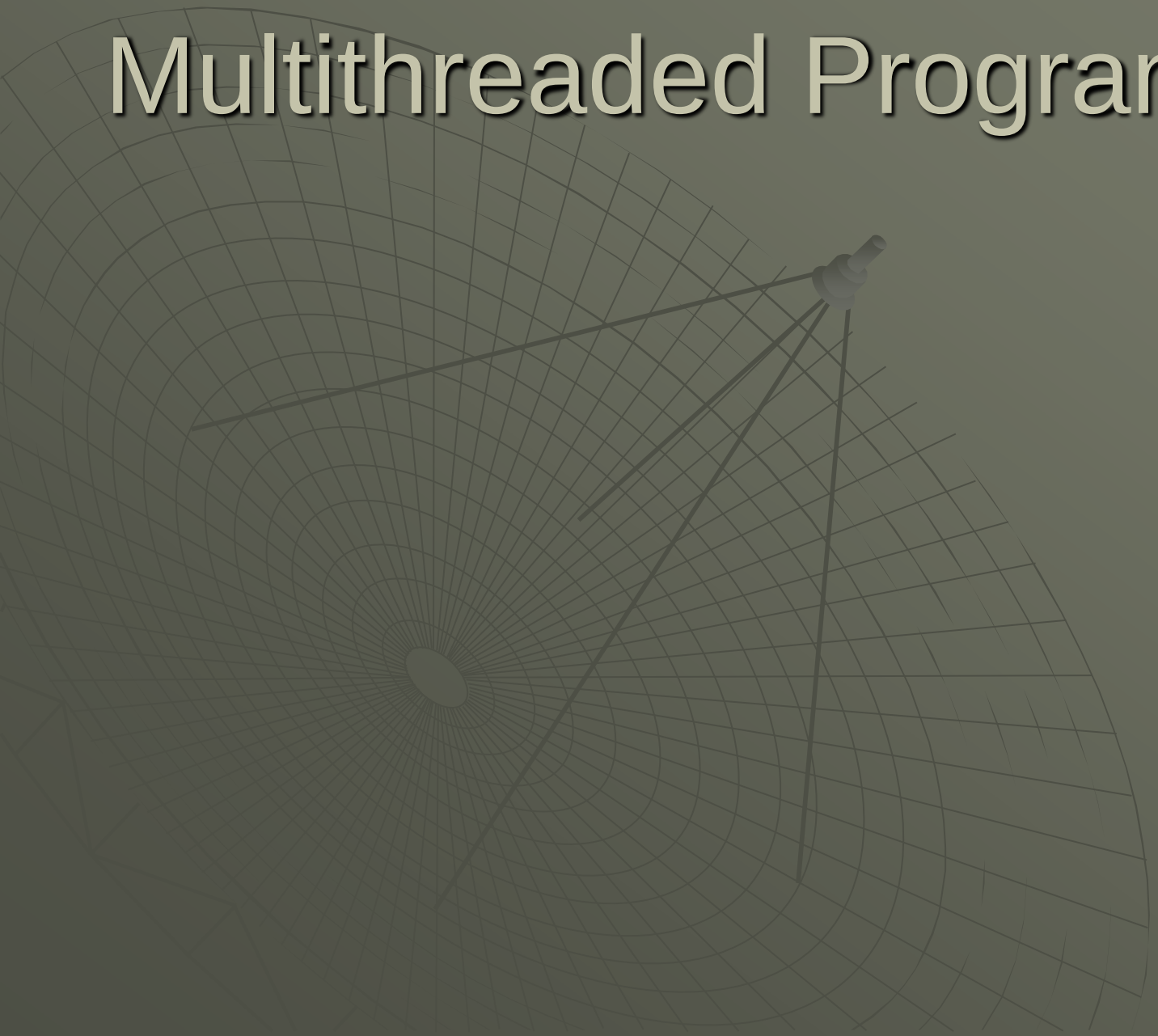
- Finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block
- Finally block will execute whether or not an exception is thrown
- If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- Finally clause is optional and each try statement requires atleast one catch or finally clause

Java’s Built-in Exceptions

- Inside the standard package **java.lang**, Java defines several exception classes.
- The most general of these exceptions are subclasses of the standard type **RuntimeException**.
- These are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions.

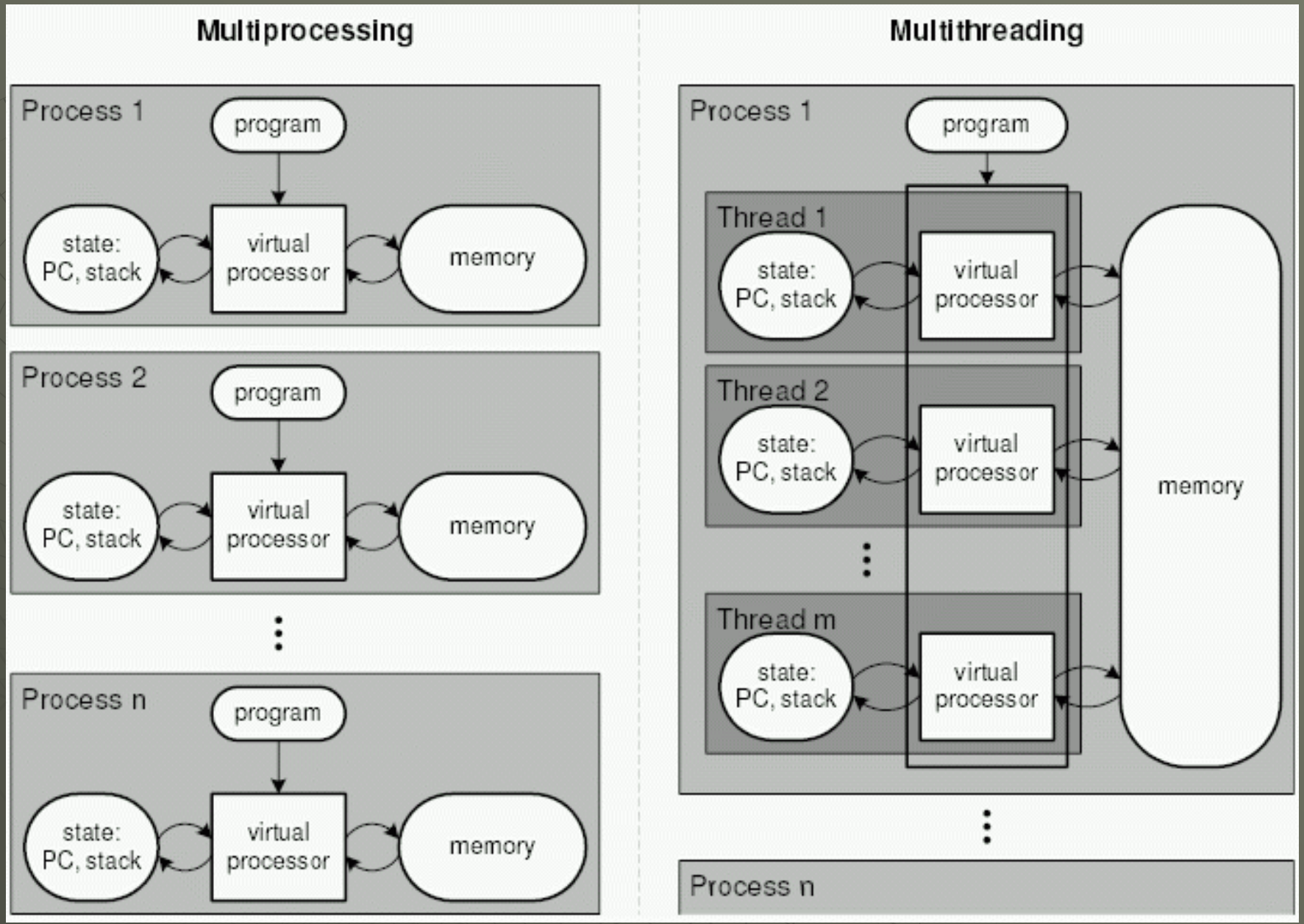
Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Multithreaded Programming



Multithreading

- Thread
  - Smallest unit of dispatchable code of a program
  - Light weight compared to a process
- Multi-Thread
  - More than one parts of a program that can execute concurrently
  - Specialized form of multitasking
  - Programmer can write code to generate multiple threads


Multiprocessing
Multithreading
Process 1
program
state:
PC, stack
virtual
processor
memory
Process 2
program
state:
PC, stack
virtual
processor
memory
Process n
program
state:
PC, stack
virtual
processor
memory
Process 1
program
Thread 1
state:
PC, stack
virtual
processor
Thread 2
state:
PC, stack
virtual
processor
Thread m
state:
PC, stack
virtual
processor
memory
Process n

Multithreading: pros and cons

- Single Threaded Program
  - allows one part of the program can execute at a time
  - It waste CPU time by keeping CPU idle. Eg: Data Read operation
  - When a thread blocks, entire program stops running
- Multithreaded Program
  - Multiple part of the program in execution, thus keeps CPU busy all time
  - Share same address space
  - Inter-Thread communication is inexpensive
  - Low cost context switch

Thread Priorities

- Integers to specify the relative priority of one thread compared to another thread
- It is used to decide when to switch from one thread to next thread (Context switch)
- Rules for context switch
  - A thread can voluntarily relinquish control
    - All other threads are examined and highest priority thread that is ready to run is given the CPU.
  - A Thread can be preempted by a higher priority Thread(preemptive multitasking)
    - Lower priority thread that does not yield the processor is simply pre-empted, no matter what is doing, by a higher priority thread.

Synchronization

- The mechanism prevents the execution of one thread, which affects the execution of another thread
- Protects shared asset being manipulated by more than one Thread at a time
- Statements in different threads executes synchronously.
- Implementation of critical region
- Java uses Monitors to handle Synchronization
- Once a Thread enters a monitor, all other threads must wait until that Thread exits the Monitor
- There is no class Monitor instead each object has its own implicit monitor that is automatically entered when one of the objects synchronized method is called.

The Main Thread

- Every Java Application program is executing with Thread called MAIN THREAD
- This Thread will automatically create when a program starts executing
- All child threads are created from this Main Thread
- This Thread terminates only after rest of child Threads are terminated
- The default name of this Thread is “main”
- When a thread object is printed, displays: [Thread Name, Priority, Group]
- Thread.currentThread(), getName(), setName(String)

- Although the main thread is created automatically when our program is started it can be controlled through a **Thread** object. To do so we must obtain a reference to it by calling the method **currentThread().**
  - **Static Thread currentThread()**
  - Returns a reference to the thread in which is called.
- setName() to change the name of a thread.
  - final void setName(string threadname)
  - final String getName()
- sleep() is used to pause the thread.
  - static void sleep(long milliseconds) throws InterruptedException

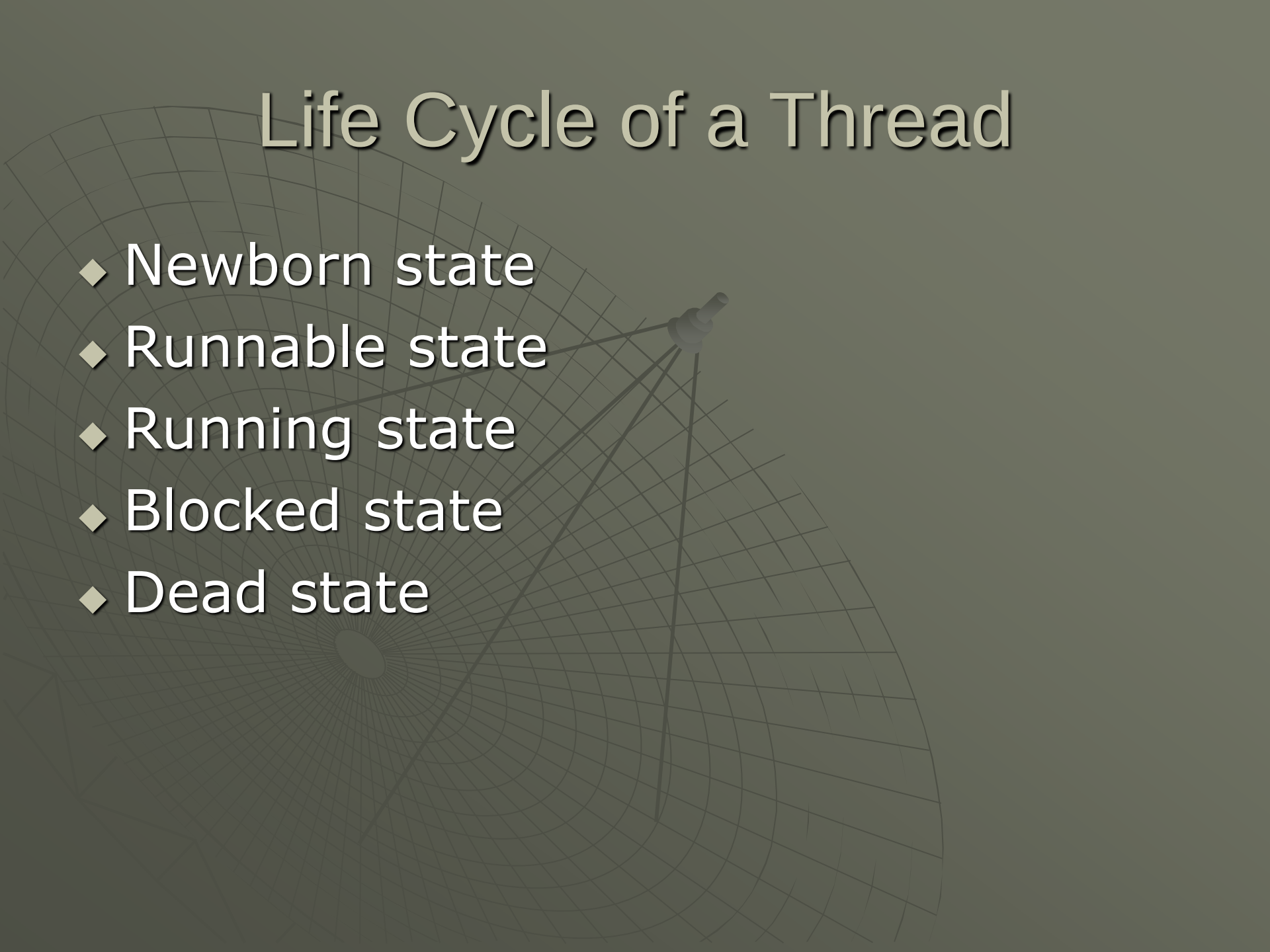
Creating Thread

- Two Methods
  - Use **Thread** Class
  - Use **Runnable** interface
- Using Thread Class
  - Create a class that extended from Thread class
  - Define a method called run() the format *public void run()*
  - Run() method contain the code that constitutes the thread
  - Run() is like any other method can do anything
  - Run() will act as the entry point for the thread and it will end when run() terminates
  - Thread is invoked by using start() method with object of class extended from Thread class
  - Example

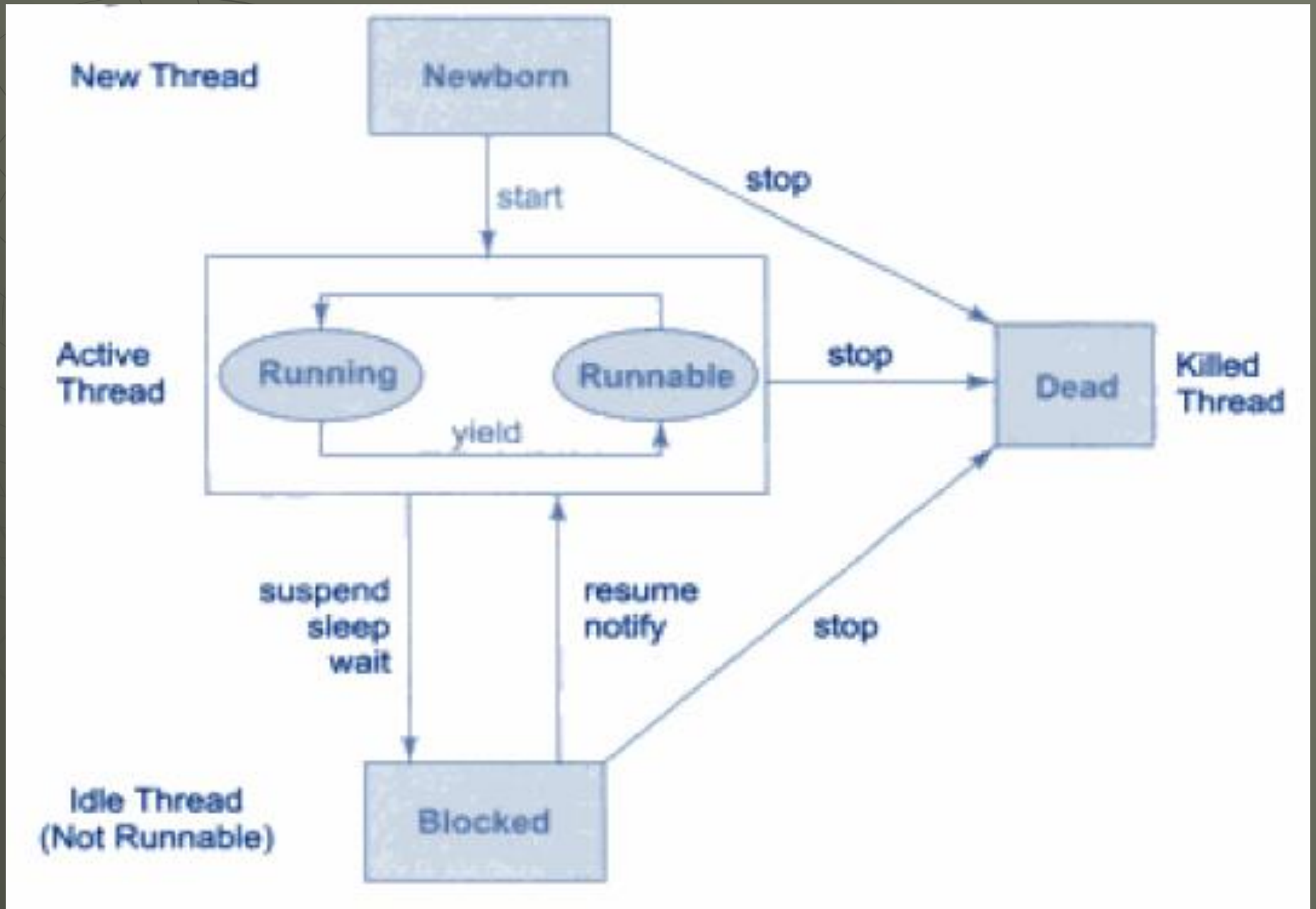
- Using Runnable Interface
  - Create a class that implements Runnable Interface
  - Redefine the abstract method run(). Eg: *public void run()*
  - Run() method contain the code that constitutes the thread
  - Run() is like any other method can do anything
  - Run() will act as the entry point for the thread and it will end when run() terminates
  - Thread is invoked by using start() method with object of class implemented Runnable interface
  - After implementing Runnable we should instantiate an object of type Thread from within that class.
    - Thread(Runnable threadOb, String threadName)
    - threadOb is an object of a class that implements Runnable interface.This defines where execution of the thread will begin.
    - Name of thread is specified by threadName.

Life Cycle of a Thread

- Newborn state
- Runnable state
- Running state
- Blocked state
- Dead state



Life Cycle of a Thread



- **Newborn state**
  - When we created a thread object, it is in newborn state.
  - We can schedule it for running using start().
  - Kill it using stop().
- **Runnable state**
  - Now it is ready for execution and waiting for the availability of the processor.
  - Enters into the thread queue.
  - time-slicing, control relinquishing and joining queue.
  - Yield() is to relinquish control to another equal priority thread before its turn comes.

- **Running state**
  - Its executing now.
  - It has been suspended using suspend() method. Using resume() we can revive a thread.
  - It has been made to sleep by calling sleep(time).
  - It has been told to wait until some event occurs using wait(). And again scheduled to run by using notify().
- **Blocked state**
  - When it is prevented from entering into the runnable state subsequently running state.

- **Dead thread**
  - Natural death after successful execution.
  - Premature death by killing a thread.

isAlive() and join()

- isAlive()
  - used to determine whether a thread has finished or not
  - Returns boolean value to indicate the status
- join()
  - Used to wait for a thread to finish
  - void join() throws InterruptedException
  - Allows to specify maximum amount of time that a Thread should wait for termination

Thread Priority

- Used by the thread scheduler to decide when each thread should be allowed to run
- Higher priority thread will get more CPU time than lower priority threads
- A lower priority thread can preempt by a higher priority thread
- Threads having equal priority should get equal access to the CPU

Implementing Thread Priority

- void setPriority(int level)
- Value of **Level** must be within the range MIN_PRIORITY (1)and MAX_PRIORITY(10)
- NORM_PRIORITY(5) used to set/get default priority
- All these are final variables in Thread class
- int getPriority() used to get the priority

Synchronization

- The mechanism used to ensure that the shared resources are accessed by more than one threads in a sequential manner
- No two threads are simultaneously operating upon a shared resource
- Synchronization is implemented by using monitor (also called Semaphore)

Monitor

- A monitor is an object that is used as a mutually exclusive lock or mutex
- Only one thread can own a monitor at a given time
- When a thread acquires a lock,
  - Thread entered the monitor
  - All other threads will suspend as long as the first thread exits the monitor
  - ie) all other threads are waiting for the monitor
  - A locked thread can reenter the same monitor

Implementing Synchronization

- Java uses Synchronized methods, methods declared with synchronized keyword
- All objects have their own implicit monitor associated with them
- Call to a synchronized method invokes the monitor in that object
- While a thread is inside the synchronized method, all other threads that try to call it on the same instance must wait
- Exiting from synchronized method releases the monitor
- Unsynchronized methods causes race condition.

The Synchronized statement

- Where to use?
  - Object of a class could access in synchronized manner
  - Parent class is a predefined class
- Solution
  - Use synchronized block to call method to be synchronized
  - General Form

```
synchronized(object)
{
    //Stmts to be synchronized
}
```

  - A synchronized block ensures that a call to a member of object occurs only after the current thread has successfully entered objects monitor

Interthread Communication

- Synchronized methods can communicate each other without using polling
- Methods to support interthreaded communication: wait(), notify(), notifyAll()
- All are final methods defined in Object class
- All methods can call from synchronized context
- wait(): tells the calling thread to give up the monitor and go to sleep until some other thread enters the same thread and calls notify()
- notify(): wakes up a thread that called wait() on the same object
- notifyAll(): wakes up all the threads that called wait() on the same object. One of the thread will granted access
- All the above methods will throw InterruptedException