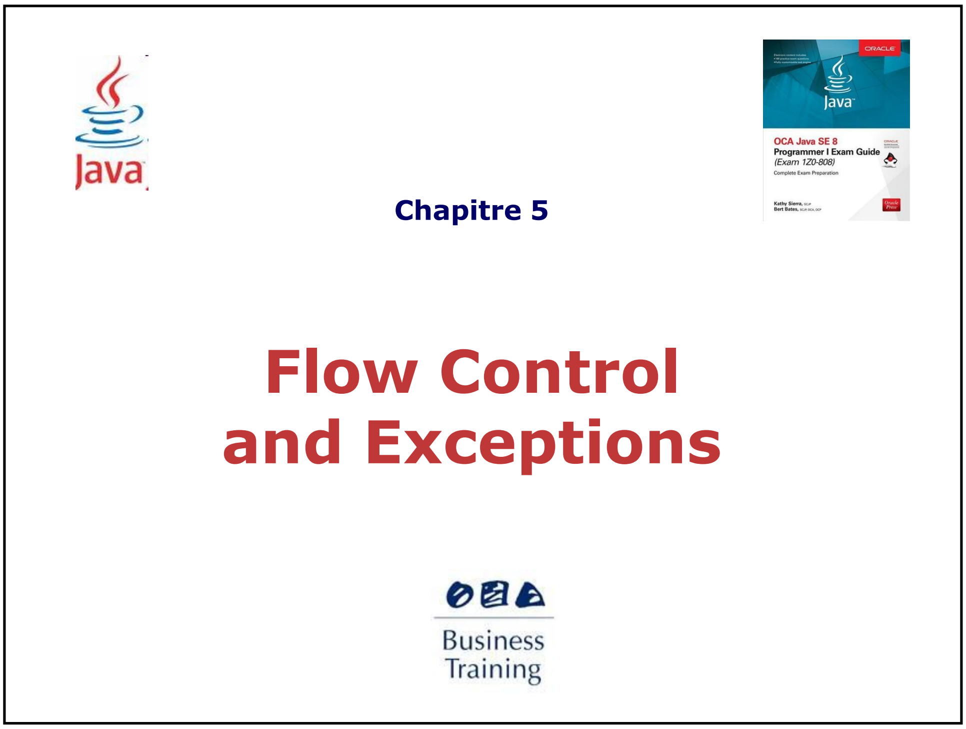
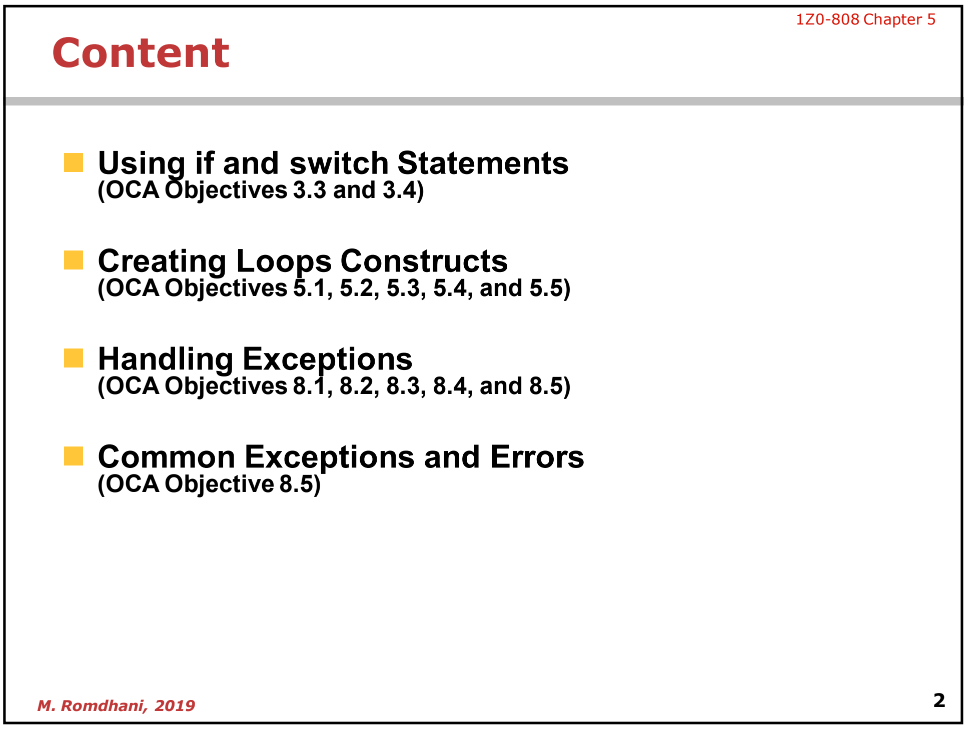


Chapitre 5

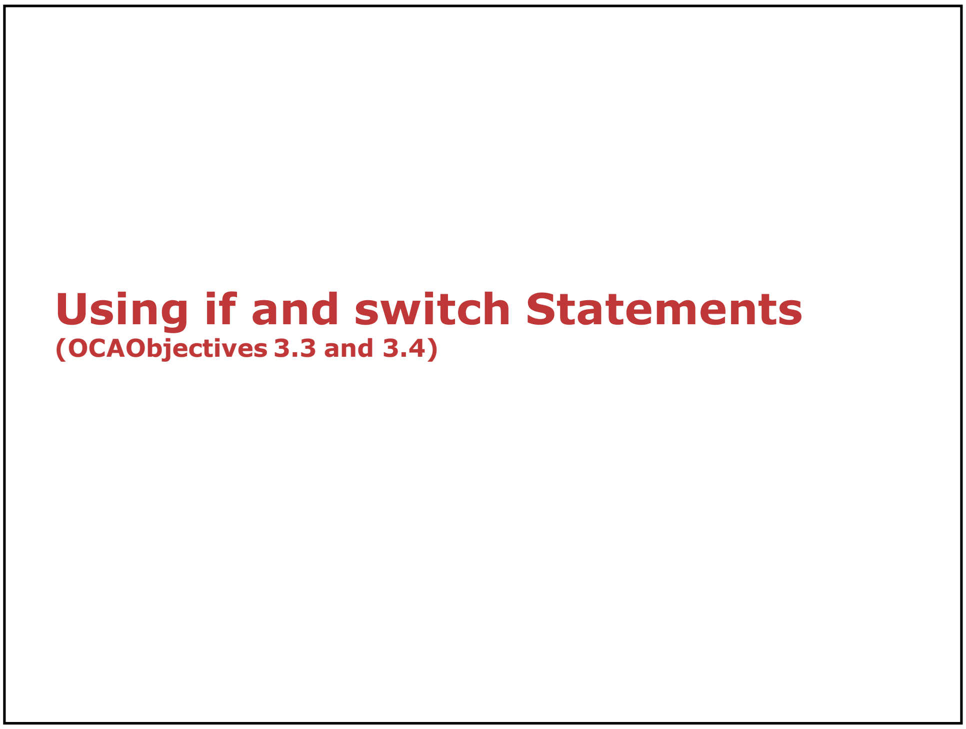
Flow Control and Exceptions

Business Training



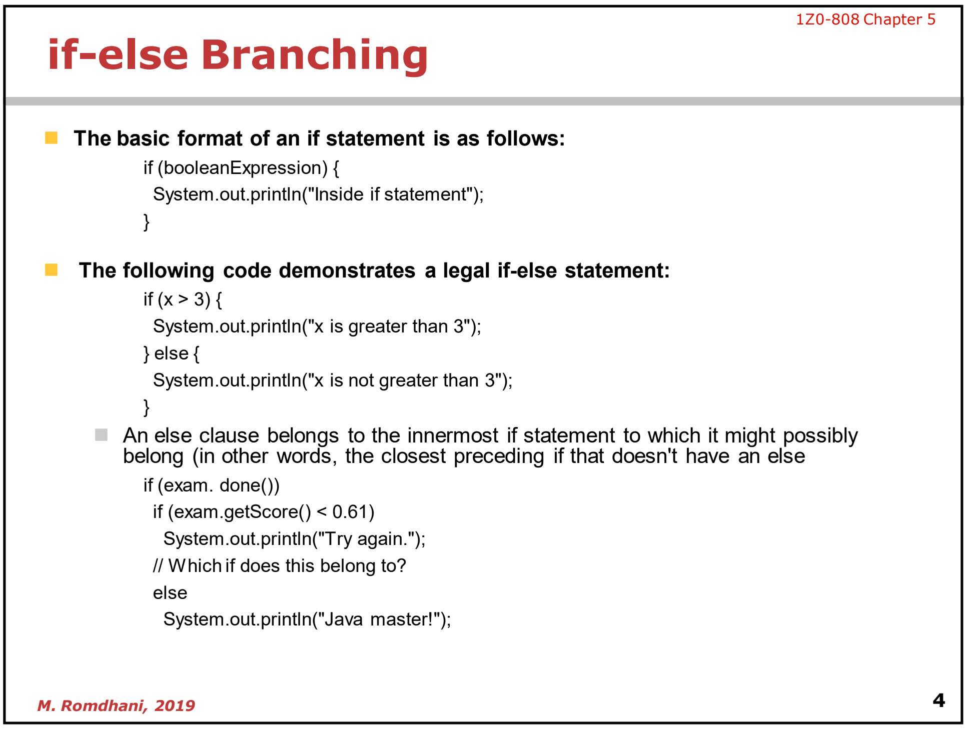
Content

- **Using if and switch Statements**
  (OCA Objectives 3.3 and 3.4)
- **Creating Loops Constructs**
  (OCA Objectives 5.1, 5.2, 5.3, 5.4, and 5.5)
- **Handling Exceptions**
  (OCA Objectives 8.1, 8.2, 8.3, 8.4, and 8.5)
- **Common Exceptions and Errors**
  (OCA Objective 8.5)



Using if and switch Statements

(OCAObjectives 3.3 and 3.4)



if-else Branching

- **The basic format of an if statement is as follows:**

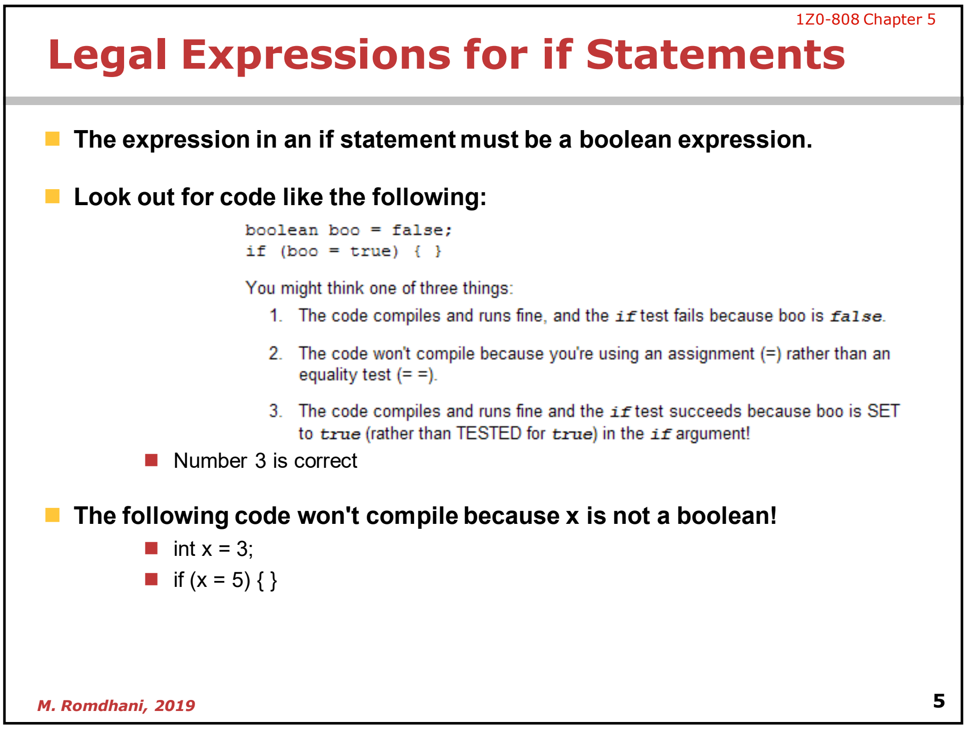
```
if (booleanExpression) {
  System.out.println("Inside if statement");
}
```

- **The following code demonstrates a legal if-else statement:**

```
if (x > 3) {
  System.out.println("x is greater than 3");
} else {
  System.out.println("x is not greater than 3");
}
```

  - An else clause belongs to the innermost if statement to which it might possibly belong (in other words, the closest preceding if that doesn't have an else

```
if (exam. done())
  if (exam.getScore() < 0.61)
    System.out.println("Try again.");
  // Which if does this belong to?
  else
    System.out.println("Java master!");
```



1Z0-808 Chapter 5

Legal Expressions for if Statements

- **The expression in an if statement must be a boolean expression.**
- **Look out for code like the following:**

```
boolean boo = false;
if (boo = true) { }
```

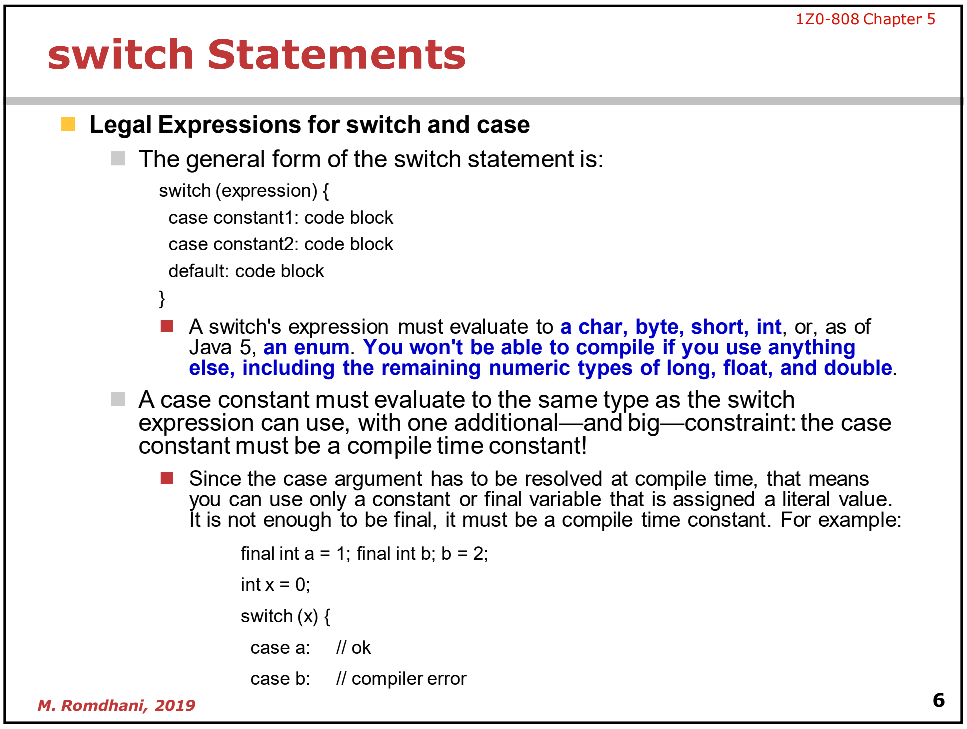
You might think one of three things:

1. The code compiles and runs fine, and the `if` test fails because boo is `false`.
2. The code won't compile because you're using an assignment (=) rather than an equality test (= =).
3. The code compiles and runs fine and the `if` test succeeds because boo is SET to `true` (rather than TESTED for `true`) in the `if` argument!

  - Number 3 is correct

- **The following code won't compile because x is not a boolean!**
  - int x = 3;
  - if (x = 5) { }

M. Romdhani, 2019 5



1Z0-808 Chapter 5

switch Statements

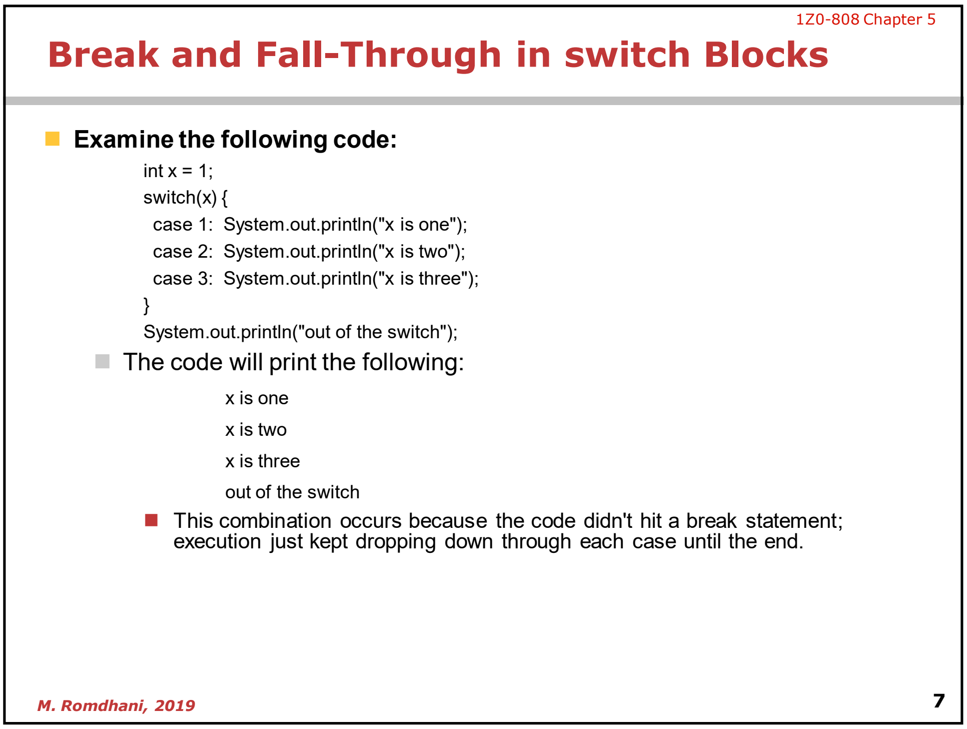
- **Legal Expressions for switch and case**
  - The general form of the switch statement is:

```
switch (expression) {
  case constant1: code block
  case constant2: code block
  default: code block
}
```

    - A switch's expression must evaluate to **a char, byte, short, int**, or, as of Java 5, **an enum**. **You won't be able to compile if you use anything else, including the remaining numeric types of long, float, and double**.
  - A case constant must evaluate to the same type as the switch expression can use, with one additional—and big—constraint: the case constant must be a compile time constant!
    - Since the case argument has to be resolved at compile time, that means you can use only a constant or final variable that is assigned a literal value. It is not enough to be final, it must be a compile time constant. For example:

```
final int a = 1; final int b; b = 2;
int x = 0;
switch (x) {
  case a:     // ok
  case b:     // compiler error
```

M. Romdhani, 2019 6



1Z0-808 Chapter 5

Break and Fall-Through in switch Blocks

- **Examine the following code:**

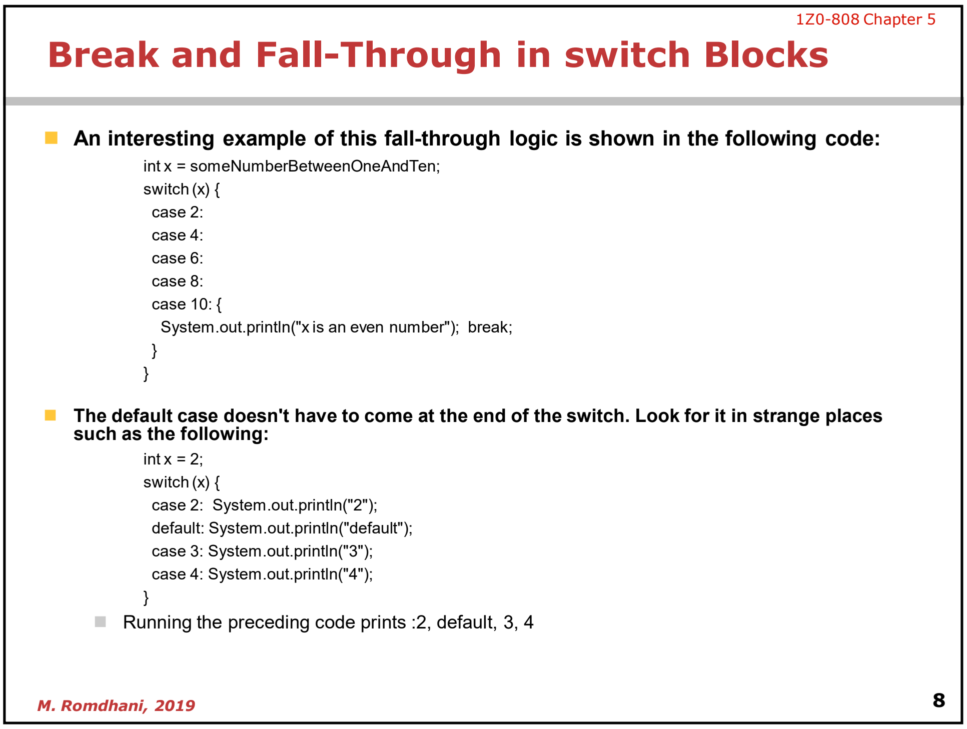
```
int x = 1;
switch(x) {
  case 1:  System.out.println("x is one");
  case 2:  System.out.println("x is two");
  case 3:  System.out.println("x is three");
}
System.out.println("out of the switch");
```

  - The code will print the following:

```
x is one
x is two
x is three
out of the switch
```

    - This combination occurs because the code didn't hit a break statement; execution just kept dropping down through each case until the end.

M. Romdhani, 2019 7



1Z0-808 Chapter 5

Break and Fall-Through in switch Blocks

- **An interesting example of this fall-through logic is shown in the following code:**

```
int x = someNumberBetweenOneAndTen;
switch (x) {
  case 2:
  case 4:
  case 6:
  case 8:
  case 10: {
    System.out.println("x is an even number");  break;
  }
}
```

- **The default case doesn't have to come at the end of the switch. Look for it in strange places such as the following:**

```
int x = 2;
switch (x) {
  case 2:  System.out.println("2");
  default: System.out.println("default");
  case 3: System.out.println("3");
  case 4: System.out.println("4");
}
```

  - Running the preceding code prints :2, default, 3, 4

M. Romdhani, 2019 8

Creating Loops Constructs

(OCA Objectives 5.1, 5.2, 5.3, 5.4, and 5.5)

Using while and do Loops

- **A while statement looks like this:**

```
while (expression) {
  // do stuff
}
```

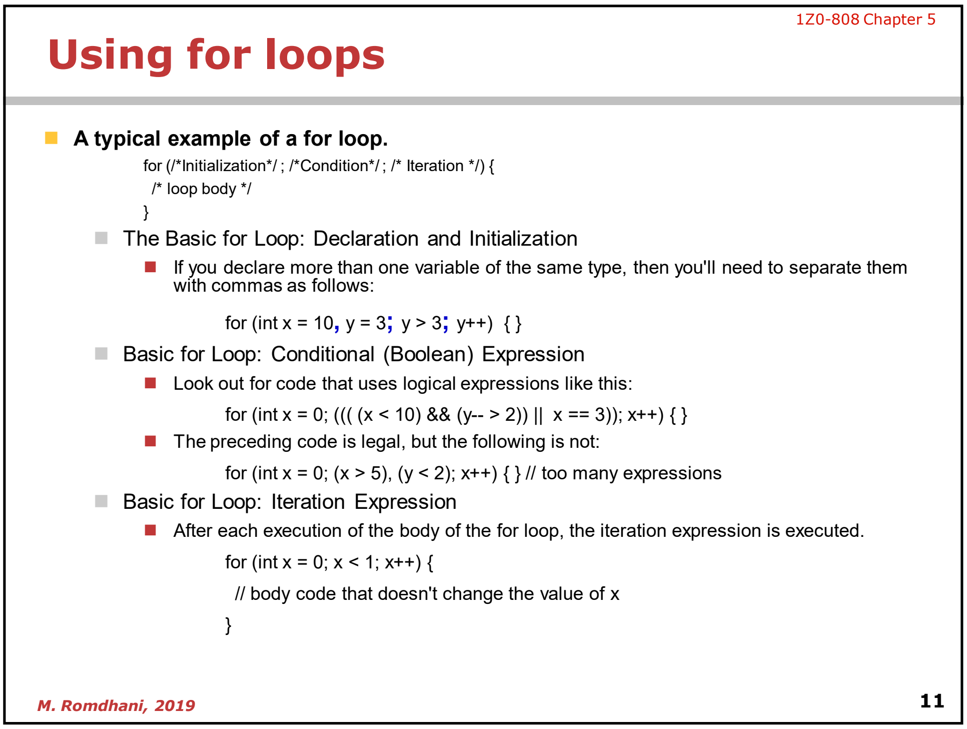
  - Any variables used in the expression of a while loop must be declared before the expression is evaluated. In other words, you can't say
    - while (int x = 2) { }   // not legal

- **The following shows a do loop in action:**

```
do {
   System.out.println("Inside loop");
} while(false);
```

- **Take a look at the following examples of legal and illegal while expressions:**

```
int x = 1;
while (x) { }         // Won't compile; x is not a Boolean
while (x = 5) { }     // Won't compile; resolves to 5
                      //(as the result of assignment)
while (x == 5) { }    // Legal, equality test
while (true) { }      // Legal
```



Using for loops

- **A typical example of a for loop.**

```
for (/*Initialization*/ ; /*Condition*/ ; /* Iteration */) {
  /* loop body */
}
```

  - The Basic for Loop: Declaration and Initialization
    - If you declare more than one variable of the same type, then you'll need to separate them with commas as follows:

```
for (int x = 10, y = 3; y > 3; y++)  { }
```

  - Basic for Loop: Conditional (Boolean) Expression
    - Look out for code that uses logical expressions like this:

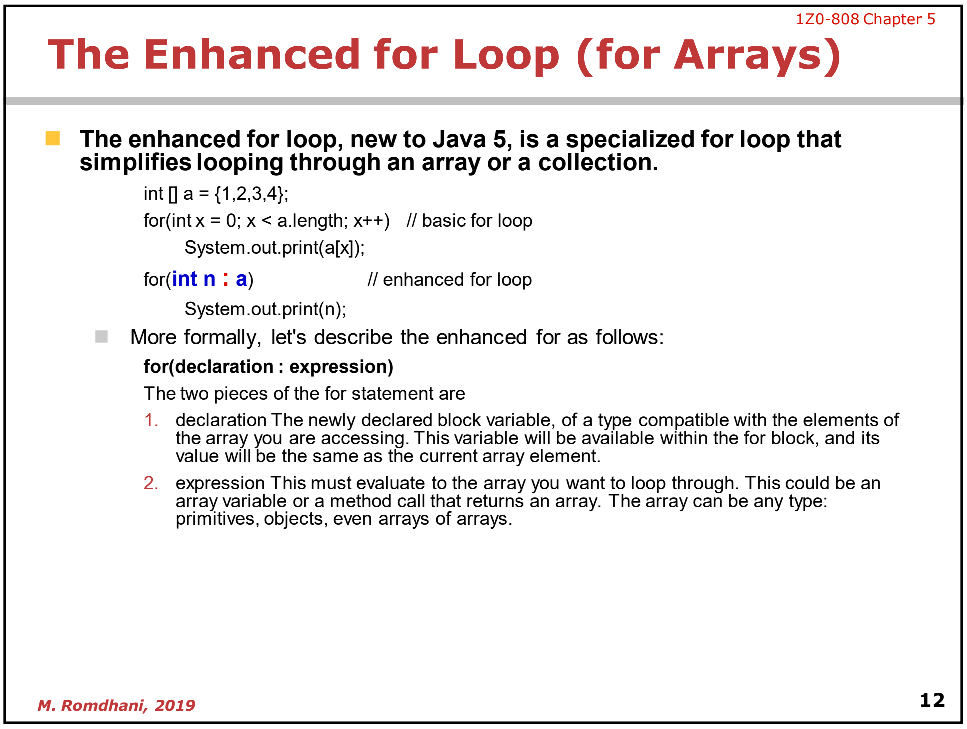
```
for (int x = 0; ((( (x < 10) && (y-- > 2)) ||  x == 3)); x++) { }
```

    - The preceding code is legal, but the following is not:

```
for (int x = 0; (x > 5), (y < 2); x++) { } // too many expressions
```

  - Basic for Loop: Iteration Expression
    - After each execution of the body of the for loop, the iteration expression is executed.

```
for (int x = 0; x < 1; x++) {
  // body code that doesn't change the value of x
}
```



The Enhanced for Loop (for Arrays)

- **The enhanced for loop, new to Java 5, is a specialized for loop that simplifies looping through an array or a collection.**

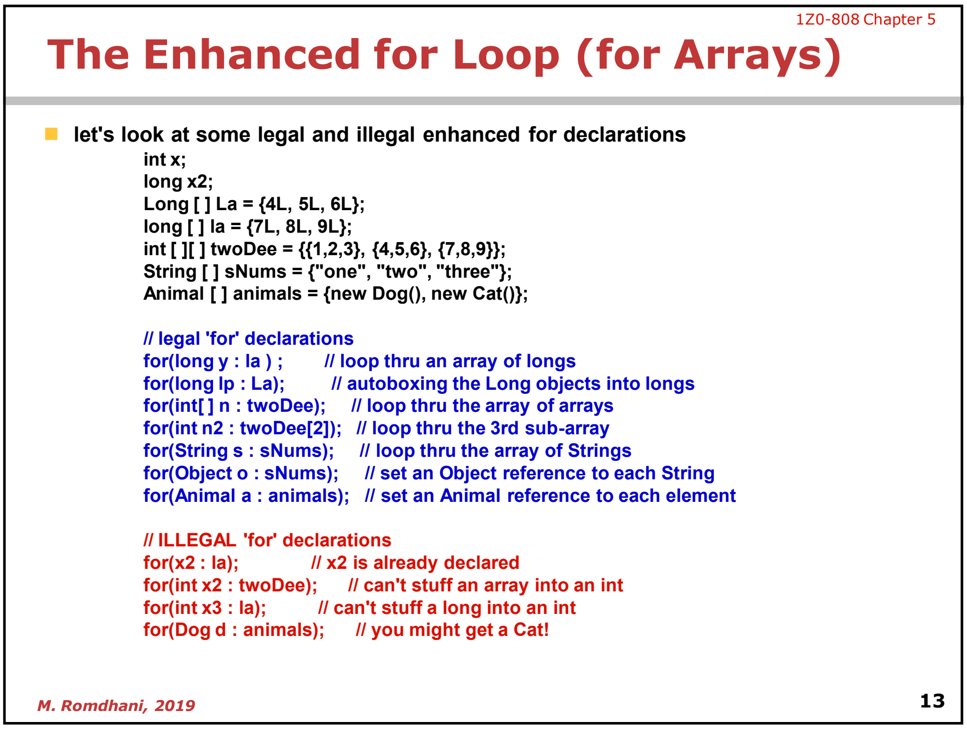
```
int [] a = {1,2,3,4};
for(int x = 0; x < a.length; x++)   // basic for loop
      System.out.print(a[x]);
for(int n : a)                      // enhanced for loop
      System.out.print(n);
```

  - More formally, let's describe the enhanced for as follows:

for(declaration : expression)

The two pieces of the for statement are

1. declaration The newly declared block variable, of a type compatible with the elements of the array you are accessing. This variable will be available within the for block, and its value will be the same as the current array element.
2. expression This must evaluate to the array you want to loop through. This could be an array variable or a method call that returns an array. The array can be any type: primitives, objects, even arrays of arrays.



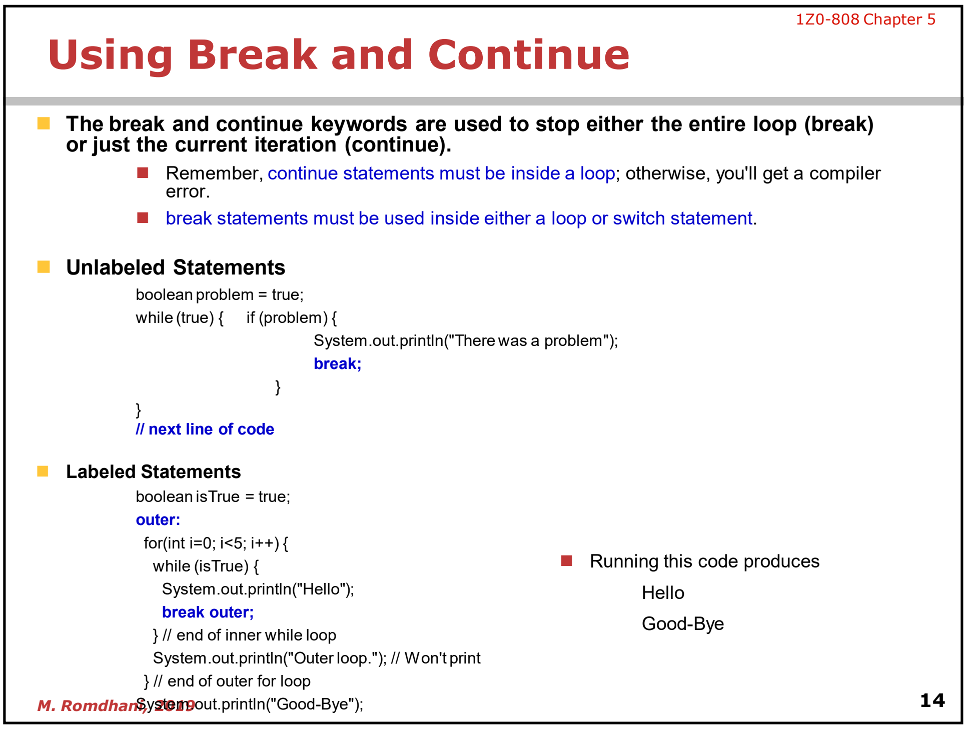
The Enhanced for Loop (for Arrays)

- **let's look at some legal and illegal enhanced for declarations**

```
int x;
long x2;
Long [ ] La = {4L, 5L, 6L};
long [ ] la = {7L, 8L, 9L};
int [ ][ ] twoDee = {{1,2,3}, {4,5,6}, {7,8,9}};
String [ ] sNums = {"one", "two", "three"};
Animal [ ] animals = {new Dog(), new Cat()};

// legal 'for' declarations
for(long y : la ) ;       // loop thru an array of longs
for(long lp : La);        // autoboxing the Long objects into longs
for(int[ ] n : twoDee);     // loop thru the array of arrays
for(int n2 : twoDee[2]);  // loop thru the 3rd sub-array
for(String s : sNums);     // loop thru the array of Strings
for(Object o : sNums);     // set an Object reference to each String
for(Animal a : animals);  // set an Animal reference to each element

// ILLEGAL 'for' declarations
for(x2 : la);              // x2 is already declared
for(int x2 : twoDee);      // can't stuff an array into an int
for(int x3 : la);          // can't stuff a long into an int
for(Dog d : animals);      // you might get a Cat!
```



Using Break and Continue

- **The break and continue keywords are used to stop either the entire loop (break) or just the current iteration (continue).**
  - Remember, continue statements must be inside a loop; otherwise, you'll get a compiler error.
  - break statements must be used inside either a loop or switch statement.
- **Unlabeled Statements**

```
boolean problem = true;
while (true) {     if (problem) {
                        System.out.println("There was a problem");
                        break;
                   }
}
// next line of code
```

- **Labeled Statements**

```
boolean isTrue = true;
outer:
 for(int i=0; i<5; i++) {
   while (isTrue) {
     System.out.println("Hello");
     break outer;
   } // end of inner while loop
   System.out.println("Outer loop."); // Won't print
 } // end of outer for loop
System.out.println("Good-Bye");
```

- Running this code produces

```
Hello
Good-Bye
```

Handling Exceptions

(OCA Objectives 8.1, 8.2, 8.3, 8.4, and 8.5)

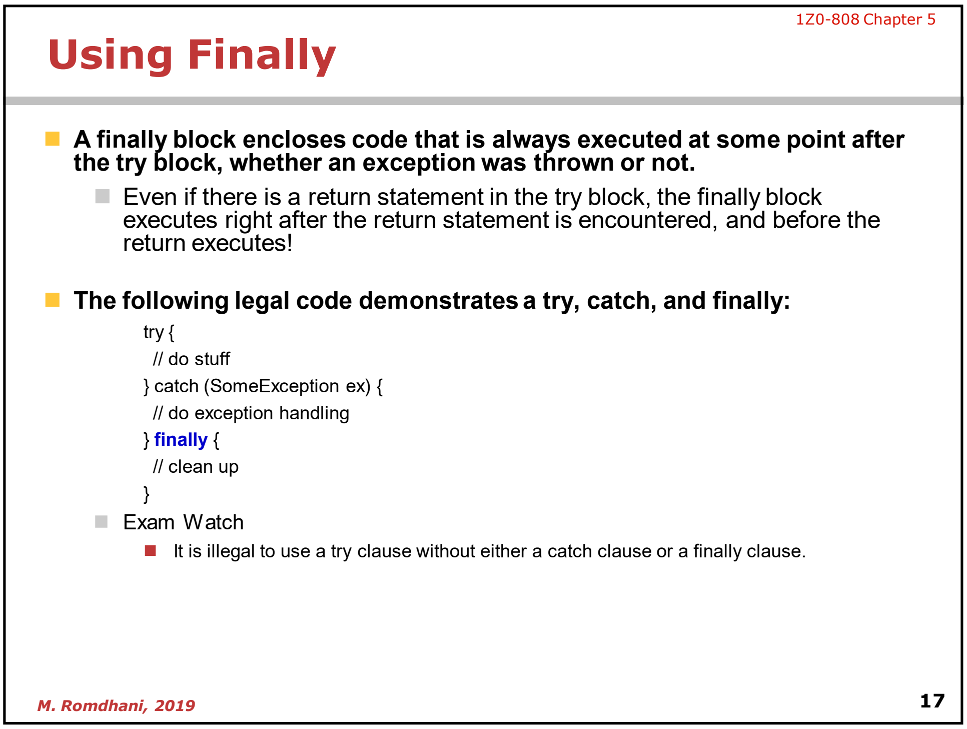
Catching an Exception Using Try and Catch

- **Here's how it looks in pseudocode:**

```
try {
  // This is the first line of the "guarded region"
  // that is governed by the try keyword.
  // Put code here that might cause some kind of exception.
  // We may have many code lines here or just one.
}
catch(MyFirstException) {
  // Put code here that handles this exception.

  // This is the next line of the exception handler.
  // This is the last line of the exception handler.
}
catch(MySecondException) {
  // Put code here that handles this exception
}

// Some other unguarded (normal, non-risky) code begins here
```

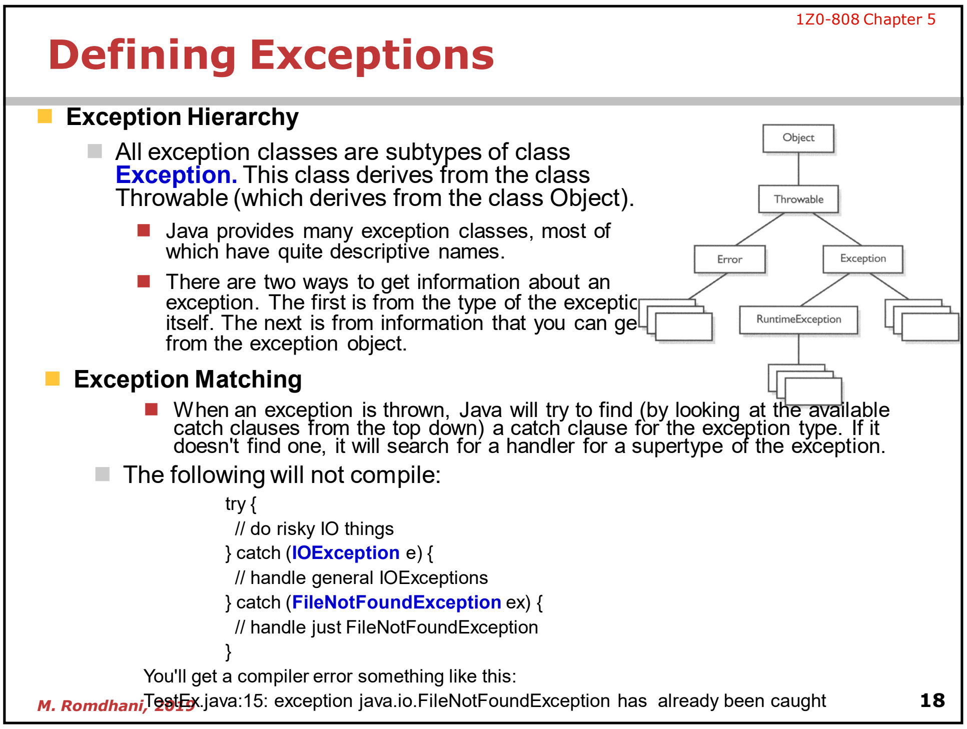


Using Finally

- **A finally block encloses code that is always executed at some point after the try block, whether an exception was thrown or not.**
  - Even if there is a return statement in the try block, the finally block executes right after the return statement is encountered, and before the return executes!
- **The following legal code demonstrates a try, catch, and finally:**

```
try {
  // do stuff
} catch (SomeException ex) {
  // do exception handling
} finally {
  // clean up
}
```

  - Exam Watch
    - It is illegal to use a try clause without either a catch clause or a finally clause.



Defining Exceptions

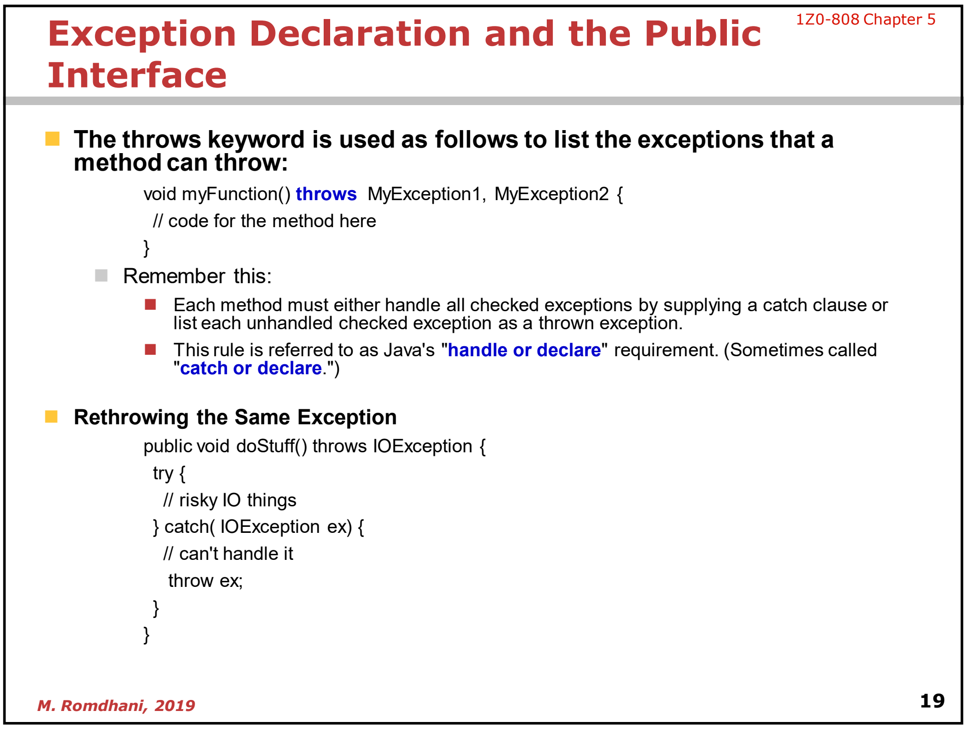
- **Exception Hierarchy**
  - All exception classes are subtypes of class **Exception.** This class derives from the class Throwable (which derives from the class Object).
    - Java provides many exception classes, most of which have quite descriptive names.
    - There are two ways to get information about an exception. The first is from the type of the exception itself. The next is from information that you can get from the exception object.

- **Exception Matching**
    - When an exception is thrown, Java will try to find (by looking at the available catch clauses from the top down) a catch clause for the exception type. If it doesn't find one, it will search for a handler for a supertype of the exception.
  - The following will not compile:

```
try {
  // do risky IO things
} catch (IOException e) {
  // handle general IOExceptions
} catch (FileNotFoundException ex) {
  // handle just FileNotFoundException
}
```

You'll get a compiler error something like this:

```
TestEx.java:15: exception java.io.FileNotFoundException has already been caught
```



Exception Declaration and the Public Interface

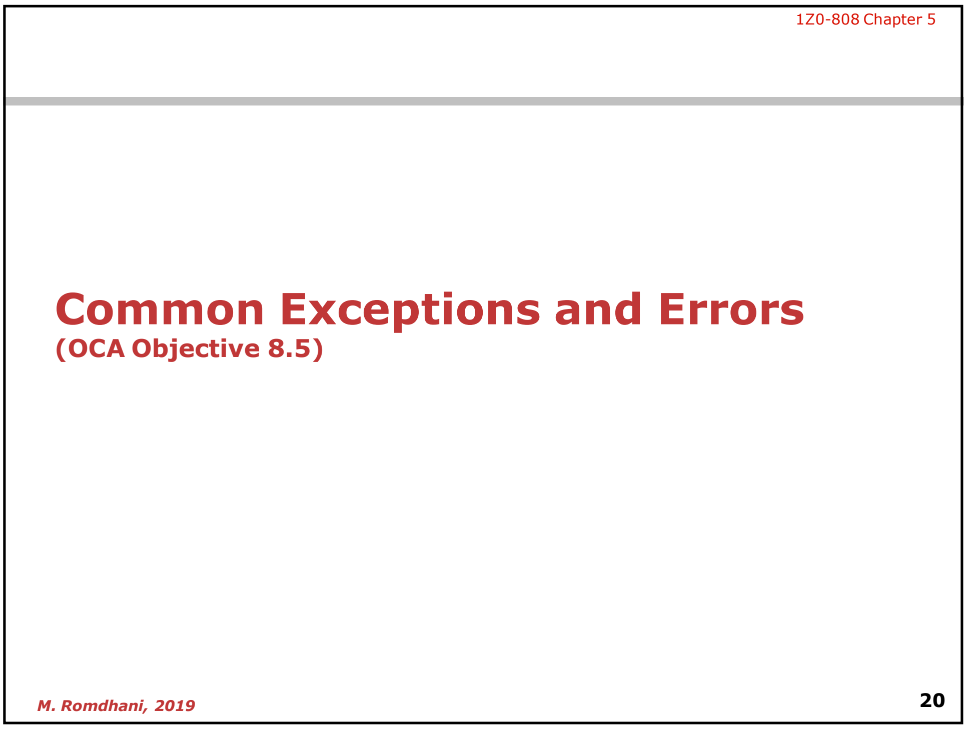
- **The throws keyword is used as follows to list the exceptions that a method can throw:**

```
void myFunction() throws  MyException1, MyException2 {
  // code for the method here
}
```

  - Remember this:
    - Each method must either handle all checked exceptions by supplying a catch clause or list each unhandled checked exception as a thrown exception.
    - This rule is referred to as Java's "**handle or declare**" requirement. (Sometimes called "**catch or declare**.")

- **Rethrowing the Same Exception**

```
public void doStuff() throws IOException {
  try {
    // risky IO things
  } catch( IOException ex) {
    // can't handle it
     throw ex;
  }
}
```



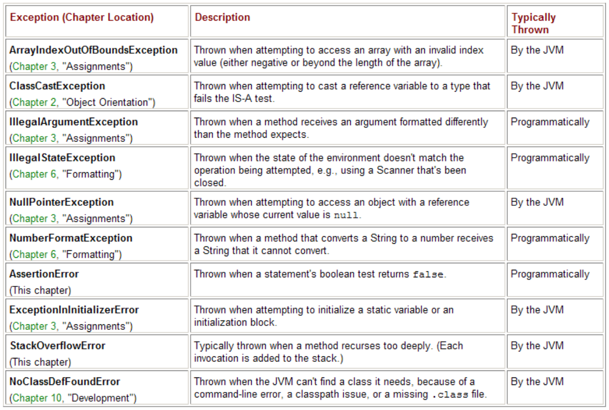
Common Exceptions and Errors
(OCA Objective 8.5)

1Z0-808 Chapter 5

Common Exceptions and Errors

- **Where Exceptions Come From ?**
  - Two broad categories of exceptions and errors:
    1. JVM exceptions Those exceptions or errors that are either exclusively or most logically thrown by the JVM.
    2. Programmatic exceptions Those exceptions that are thrown explicitly by application and/or API programmers.

- **Descriptions and Sources of Common Exceptions**



Exception (Chapter Location)	Description	Typically Thrown
ArrayIndexOutOfBoundsException (Chapter 3, "Assignments")	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array).	By the JVM
ClassCastException (Chapter 2, "Object Orientation")	Thrown when attempting to cast a reference variable to a type that fails the IS-A test.	By the JVM
IllegalArgumentException (Chapter 3, "Assignments")	Thrown when a method receives an argument formatted differently than the method expects.	Programmatically
IllegalStateException (Chapter 6, "Formatting")	Thrown when the state of the environment doesn't match the operation being attempted, e.g., using a Scanner that's been closed.	Programmatically
NullPointerException (Chapter 3, "Assignments")	Thrown when attempting to access an object with a reference variable whose current value is `null`.	By the JVM
NumberFormatException (Chapter 6, "Formatting")	Thrown when a method that converts a String to a number receives a String that it cannot convert.	Programmatically
AssertionError (This chapter)	Thrown when a statement's boolean test returns `false`.	Programmatically
ExceptionInInitializerError (Chapter 3, "Assignments")	Thrown when attempting to initialize a static variable or an initialization block.	By the JVM
StackOverflowError (This chapter)	Typically thrown when a method recurses too deeply. (Each invocation is added to the stack.)	By the JVM
NoClassDefFoundError (Chapter 10, "Development")	Thrown when the JVM can't find a class it needs, because of a command-line error, a classpath issue, or a missing `.class` file.	By the JVM

M. Romdhani, 2019