Correctness and Robustness

With correctness, problems are the fault of the program's (or module's) code.

- only need to check for the correct situation
- program code only needs to detect a problem
  - handling the problem (i.e. fixing the bug) is the programmer's job

With robustness, problems are the fault of something outside the program (or module).

- check for incorrect situations
- program code also needs to handle the problem
  - attempt to recover if possible – e.g. prompt for new input
  - if recovery is not possible, avoid doing damage and display an informative message so the user can understand and attempt to fix the problem
  - crashing is not reasonable

Implementing Robustness

Considerations –

- can the error be handled in the same place it is detected?

```
int number;
for ( ; true ; ) {
  System.out.println("Enter a positive "+
                     "number: ");
  number = scanner.nextInt();
  if ( number > 0 ) { break; }
  else {
    System.out.println("not positive!");
  }
}
System.out.println("the square root is "+
                   Math.sqrt(number));
```

  - this code reads the original value from the user, so it knows that reading a new value from the user is appropriate if the user doesn't enter a positive number

```
/**
 * …
 * @param x … (x > 0)
 */

public void func ( int x ) {
  if ( x <= 0 ) { … }
  …
}
```

  - func has no idea where the parameter value came from, so it cannot take appropriate actions to remedy the bad value – it can only signal that an error occurred so the caller can do something

Signaling Errors

Two options –

```
int index = str.indexOf(',');
if ( index == -1 ) { … }   // handle error
String before = str.substring(0,index);
```

- return a special value
  - error handling is part of the normal flow of control
  - not always possible
  - caller can ignore
- throw an exception
  - *typically only used when the error truly is an error, not a normally expected outcome*
  - can separate error handling from the normal flow of control
  - allows for streamlined error handling
    - simplify code by grouping error handing for a related block of code in one place
    - simplify code when execution of a block of code should not continue if there's a problem with one step
  - some exception types must be handled

Throwing Exceptions

```
if ( condition ) {
  throw exception-object;
}
```

The thing being thrown is an object.

- typically you create a new object rather than throwing one that already exists
- contains info about what went wrong and where
  - object type provides information about the problem which can be used in the program to determine how to handle the problem
  - detail message provides more information for the programmer/user
  - stack trace identifies where the exception was thrown

```
e.getMessage();
e.printStackTrace();
```

Example

```
/**
 * pre: low <= high
 */
public void func ( int low, int high ) {
  if ( low > high ) {
    throw new IllegalArgumentException("require "+
              " low <= high; got "+low+" > "+high);
  }
  …
}
```

for preconditions, the convention in Java is to use `IllegalArgumentException`


```
public static Card read () throws ParseException {
  int value = TextIO.getInt();
  if ( value < 1 || value > 13 ) {
    throw new ParseException("value must be between 1 "+
                             " and 13; got "+value);
  }
  char ch = TextIO.getAnyChar();
  if ( ch != ' ' ) {
    throw new ParseException("expected space; got "+ch);
  }
  String word = TextIO.getWord();
  if ( !word.equals("of") ) {
    throw new ParseException("expected 'of'; got "+
                             value);
  }
  char ch2 = TextIO.getAnyChar();
  if ( ch2 != ' ' ) {
    throw new ParseException("expected space; got "+ch2);
  }
  String suit = TextIO.getWord();
  if ( !suit.equals("hearts") && !suit.equals("diamonds") &&
       !suit.equals("spades") && !suit.equals("clubs") ) {
    throw new ParseException("invalid suit "+suit);
  }
  return new Card(suit,value);
}
```


Types of Exceptions

A class hierarchy groups types of exceptions by function.

- `Throwable` is the top-level class
  - two main subclasses: `Error` and `Exception`
- `Error` is used for serious, fatal errors – something that there is no reasonable way to handle
  - e.g. problems with the Java VM
  - should not be caught – nothing the program can do
- `Exception` is used for errors that can be handled
  - generally the only things you'll throw (and catch) are subclasses of `Exception`
  - must be caught (unless subclass of `RuntimeException`)

Types of Exceptions

- `RuntimeException` is a subclass of `Exception`
  - used to signal failed runtime checks
    - e.g. `ArrayIndexOutOfBoundsException`, `NullPointerException`, `IllegalArgumentException`
  - generally the only one you'd throw is `IllegalArgumentException`
  - generally should not be caught – fix the bug instead

Question	"expected" errors that can be anticipated and should typically be handled - catch	"unrecoverable" errors - do not catch	typically indicate bugs - do not catch
Exception (but not RuntimeException)	✓ 12	2	2
Error	2	✓ 10	4
RuntimeException	3	2	✓ 11

Types of Exceptions

Choose an appropriate exception type for your error.

- Java provides a number of exception types (though most are for very specific purposes)
- can also define your own
  - subclass `Exception`, `RuntimeException`, or an existing exception type

Direct Known Subclasses:
AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CertificateException, ClassNotFoundException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalAccessException, IllegalClassFormatException, InstantiationException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, InvocationTargetException, IOException, JAXBException, JMException, KeySelectorException, LastOwnerException, LineUnavailableException, MarshalException, MidiUnavailableException, MimeTypeParseException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NoSuchFieldException, NoSuchMethodException, NotBoundException, NotOwnerException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ScriptException, ServerNotActiveException, SOAPException, SQLException, TimeoutException, TooManyListenersException, TransformerException, TransformException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, UserException, XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException

Direct Known Subclasses:
AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, ConcurrentModificationException, DataBindingException, DOMException, EmptyStackException, EnumConstantNotPresentException,

Catching Exceptions

A section of code that may throw an exception goes inside a *try block*.

One or more *catch blocks* (one for each type of exception) contain code to be executed if that type of exception is thrown.

Catching Exceptions

When an exception is thrown, control immediately transfers to the nearest enclosing catch block.

- matching means the thrown object's type is the same or a subclass of the catch block's declared exception type
- if there is no matching catch within the current method, a matching catch is sought in the caller (and so forth)

Then –

- the body of the catch block is executed
- the `finally` block is executed (if any)
- control continues immediately following that try-catch

Example

10, 20, abc, 30, 0, 40, and xyz

```
int sum = 0;
for ( ; true ; ) {

  String input;
  System.out.print("please type something: ");
  input = scanner.nextLine();
  int number = Integer.parseInt(input);

  if ( number == 0 ) { break; }
  sum = sum+number;

}

System.out.println("the sum is: "+sum);
```


Example

10, 20, abc, 30, 0, 40, and xyz

```
int sum = 0;
for ( ; true ; ) {

  String input;
  System.out.print("please type something: ");
  input = scanner.nextLine();

  try {
    int number = Integer.parseInt(input);
    if ( number == 0 ) { break; }
    sum = sum+number;

  } catch ( NumberFormatException e ) {
    System.out.println("that wasn't a number!");
  }
}

System.out.println("the sum is: "+sum);
```


Example

10, 20, abc, 30, 0, 40, and xyz

```
int sum = 0;

try {
  for ( ; true ; ) {

    String input;
    System.out.print("please type something: ");
    input = scanner.nextLine();
    int number = Integer.parseInt(input);
    if ( number == 0 ) { break; }
    sum = sum+number;
  }

  System.out.println("the sum is: "+sum);

} catch ( NumberFormatException e ) {
  System.out.println("that wasn't a number!");
}
```


Example

10, 20, abc, 30, 0, 40, and xyz

```
int sum = 0;

try {
  for ( ; true ; ) {

    String input;
    System.out.print("please type something: ");
    input = scanner.nextLine();
    int number = Integer.parseInt(input);
    if ( number == 0 ) { break; }
    sum = sum+number;
  }
} catch ( NumberFormatException e ) {
  System.out.println("that wasn't a number!");
} finally {
  System.out.println("the sum is: "+sum);
}
```

Example

10, 20, abc, 30, 0, 40, and xyz

```
int sum = 0;

try {
  for ( ; true ; ) {

    String input;
    System.out.print("please type something: ");
    input = scanner.nextLine();
    int number = Integer.parseInt(input);
    if ( number == 0 ) { break; }
    sum = sum+number;
  }
} catch ( NumberFormatException e ) {
  System.out.println("that wasn't a number!");
}

System.out.println("the sum is: "+sum);
```



```
int[] numbers = new int[10];
for ( int index = 0 ; index < numbers.length ;
      index++ ) { numbers[index] = index; }

for ( ; true ; ) {

  String input;
  System.out.print("please type something: ");
  input = scanner.nextLine();

  try {
    int number = Integer.parseInt(input);
    if ( number == 0 ) { break; }
    System.out.println(numbers[number]);

  } catch ( NumberFormatException e ) {
    System.out.println("that wasn't a number!");
  } catch ( ArrayIndexOutOfBoundsException e ) {
    System.out.println("not a legal index!");
  }
}
```



```
int[] numbers = new int[10];
for ( int index = 0 ; index < numbers.length ;
      index++ ) { numbers[index] = index; }

try {
  for ( ; true ; ) {

    String input;
    System.out.print("please type something: ");
    input = scanner.nextLine();

    try {
      int number = Integer.parseInt(input);
      if ( number == 0 ) { break; }
      System.out.println(numbers[number]);

    } catch ( NumberFormatException e ) {
      System.out.println("not a number!");
    }
  }
} catch ( ArrayIndexOutOfBoundsException e ) {
    System.out.println("not a legal index!");
}
```


When should you use a `finally` block? Choose all that apply.

	Answer	Respondents	Percentage
✓	To ensure that resources such as files, network connections, or database connections are closed properly, even if an exception occurs.	14	32%
×	To catch exceptions and prevent the program from terminating due to an unhandled exception.	2	5%
✓	To execute cleanup code that must run regardless of whether an exception is thrown, such as resetting variables, deleting temporary files, or releasing locks.	15	34%

the catch block is for error handling – the finally block is executed regardless of whether or not an exception occurred in the try block

When should you use a `finally` block? Choose all that apply.

×	To replace the need for a catch block, since finally will always execute.	1	2%
×	To guarantee that a certain piece of code is executed before the method exits, even if the method contains multiple return statements.	12	27%

the catch block is for error handling – the finally block is executed regardless of whether or not an exception occurred in the try block

technically correct but not the intent of `finally`

```
public static void main ( String[] args ) throws Exception {

  try {
    System.out.println('A');

    try {
      System.out.println('B');
      return;

    } finally {
      System.out.println('X');
    }

  } catch ( Exception e ) {
    System.out.println('Y');

  } finally {
    System.out.println('Z');
  }
}
```

A
B
X
Z

Caller's Responsibility

When the problem cannot be handled in the place it was detected, the caller has two options for handling –

- look before you leap
  - check for problematic conditions and only call the method if it can succeed
- leap before you look
  - call the method and deal with the error when it is signaled

Generally look first if you can. Prefer leaping if –

- determining if the error condition exists is difficult
- errors are signaled with exceptions (but not `RuntimeExceptions`)

leap before you look because determining if `str` has a comma requires looking for a comma in `str`...

```
int index = str.indexOf(',');
if ( index == -1 ) { … }   // handle error
String before = str.substring(0,index);
```

Not Catching Exceptions

You may choose not to catch/handle an exception in a method.

- e.g. the method can't solve the problem, but its caller might be able to

A method that throws an exception (other than a `RuntimeException`) without catching it must indicate that.

- `@exception` tag in javadoc comments identifying exception type and when it occurs
- `throws` declaration in method header

Example

```
/**
 * …
 * @exception FileNotFoundException
 *   if specified file does not exist
 * @exception IOException
 *   if an I/O error occurs while reading the file
 */
public void load ( String filename )
  throws FileNotFoundException, IOException {

  BufferedReader reader =
    new BufferedReader(new FileReader(filename));

  for ( ; true ; ) {
    String line = reader.readLine();
    if ( line == null ) { break; }
    …
  }
}
```

Appropriate Use of Exceptions

Exceptions are handy, but they aren't for everything.

- not appropriate for when the error can (and should) be handled right away

```
System.out.println("enter a number 1-10: ");
int number = scanner.nextInt();
for ( ; number < 1 || number > 10 ; 0 {
  System.out.println("number must be between "+
                     "1 and 10");
  number = scanner.nextInt();
}
```



```
int[] numbers = new int[10];
for ( int index = 0 ; index < numbers.length ;
      index++ ) { numbers[index] = index; }

for ( ; true ; ) {

  String input;
  System.out.print("please type something: ");
  input = scanner.nextLine();

  try {
    int number = Integer.parseInt(input);
    if ( number == 0 ) { break; }
    System.out.println(numbers[number]);

  } catch ( NumberFormatException e ) {
    System.out.println("that wasn't a number!");
  } catch ( ArrayIndexOutOfBoundsException e ) {
    System.out.println("not a legal index!");
  }
}
```

catching a RuntimeException isn't generally the best style – what should you do instead?

In which of the following situations would it be most appropriate to throw an exception instead of handling the issue directly or returning a special value? Choose all that apply.

	Answer	Respondents	Percentage
✓	When a method that expects a positive number for a parameter instead receives a negative number.	8	17%
✓	When a method receives a filename as a parameter but the file doesn't exist.	12	26%
×	When a user enters invalid or unrecognized input, such as entering a negative number when a positive number is asked for.	10	22%

when the user has entered something, the correct handling of bad input is known – have the user enter a new something

In which of the following situations would it be most appropriate to throw an exception instead of handling the issue directly or returning a special value? Choose all that apply.

×	When a method to sort an array receives an already-sorted array, since sorting is unnecessary in this case.	2	4%
✓	When a method to remove an element from a collection is called on an empty collection.	10	22%
×	When a method to find a substring in a string is called with a substring that doesn't exist in the string, such as find("xyz","zebra").	4	9%

this is not an error – the method's job is done! the array is sorted!

"not found" is a normal, expected outcome for find – exceptions are meant for problems