CSC 1052 Algorithms & Data Structures II

Inheritance, More Algorithm Efficiency, Exceptions and Collections

INHERITANCE

Inheritance

Inheritance is the object-oriented programming technique that allows one class to be derived from another

For example, if we already have a BankAccount class, we could use it to derive a SavingsAccount class

The SavingsAccount class would inherit the attributes and behaviors of a BankAccount

We could then add elements that make a savings account different than other bank accounts

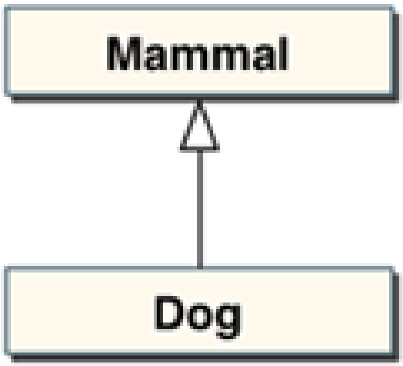
Inheritance lets us reuse common characteristics in a similar class without redefining them

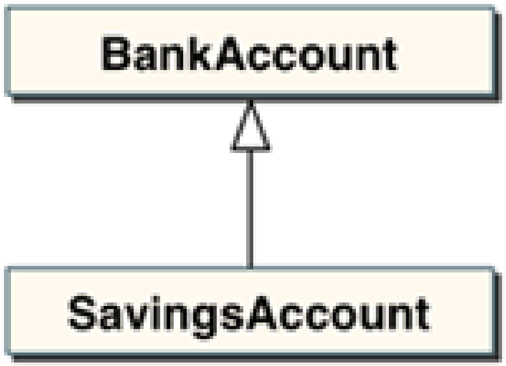
Inheritance

The existing class is called the superclass (or parent class, or base class)

The derived class is called the subclass (or child class)

An inheritance relationship is depicted in a UML class diagram with an arrow pointing toward the superclass





Inheritance

Inheritance should define an is-a relationship

The subclass is a more specific version of the superclass

- All dogs are mammals, but not all mammals are dogs
- A savings account is a bank account that earns interest

This is the basic idea behind classification schemes

Inheritance

Recall that the BankAccount class manages an account number and a balance

It has operations for getting the current balance and making deposits and withdraws

To derive the SavingsAccount class, we use an extends clause:

```
public class SavingsAccount extends BankAccount
{
    // content specific to SavingsAccount goes here
}
```

Inheritance

By extending BankAccount, a SavingsAccount already has an account number and a balance

The interest rate can be added to the SavingsAccount class

Constructors are not inherited, but the superclass constructor can be called using the super reference

```
public SavingsAccount(long accountNumber,
    double balance, double interestRate)
{
    super(accountNumber, balance);
    this.interestRate = interestRate;
}
```

Inheritance

We can add a method in SavingsAccount called addInterest to compute the interest and add it to the balance

```
public double addInterest()
{
    double interest = getBalance() * interestRate / 100;
    return deposit(interest);
}
```

The getBalance method and deposit method are inherited from BankAccount

Inheritance

Sometimes the parent's version of a method is not adequate, so the child class overrides it by providing its own definition

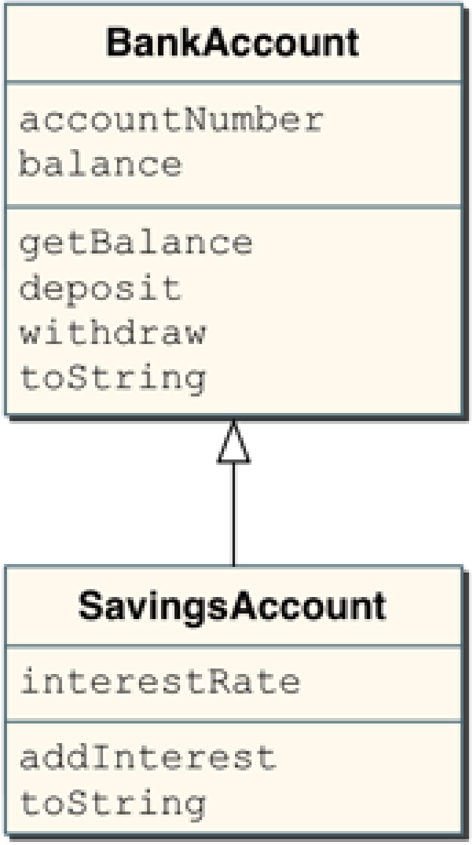
The child class may use the super reference to explicitly call the parent's version of a method

```
public String toString()
{
    String description = super.toString();
    description += String.format(" (earns %3.1f%% interest)",
        interestRate * 100);

    return description;
}
```

Inheritance

A class diagram may show details of the relationship between classes



Class Hierarchies

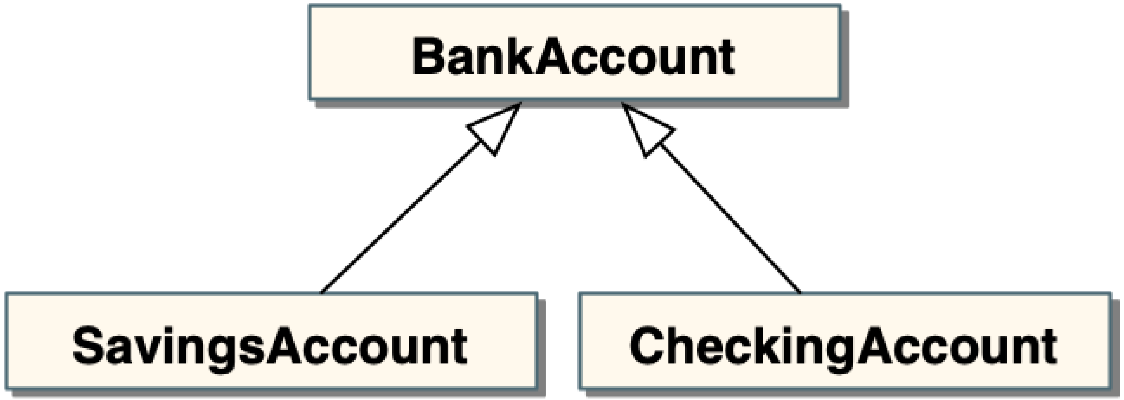
The idea of Inheritance is that one class can be derived from another. In this way the data and methods of the parent class are inherited by the child class.

In Java, a class has only one parent, but can have multiple children.

There is no limit to the number of subclasses a class may have.

Class Hierarchies

In this example, the BankAccount class is the parent of both the SavingsAccount and CheckingAccount classes.

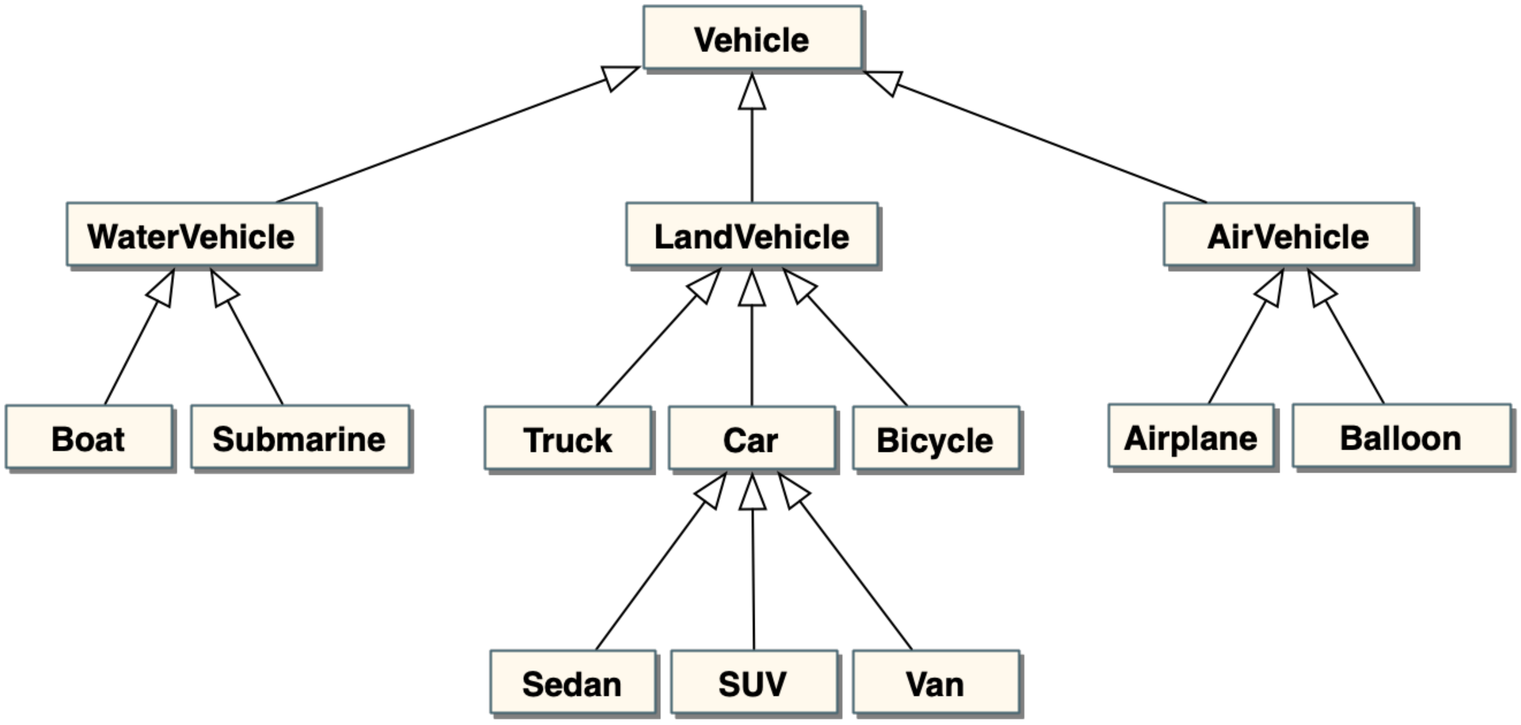


The data and methods of the BankAccount class are inherited by both of the subclasses.

Any two children of the same parent are called sibling classes.

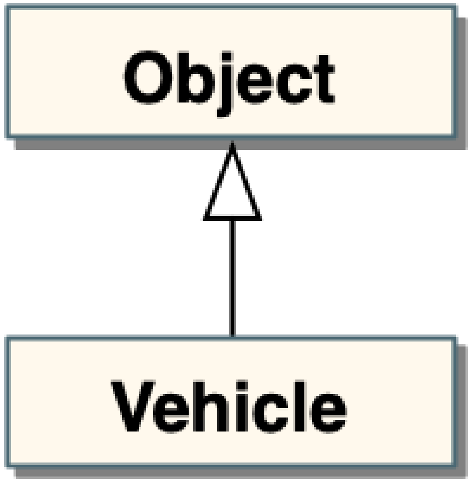
Class Hierarchies

A child of one class can be the parent of another, such that inheritance relationships form class hierarchies.



Class Hierarchies

In Java, when a class does not explicitly inherit from a parent class it inherits from the Object class.



On the previous slide, the Vehicle class is the root of the hierarchy. All classes in the hierarchy inherit its elements.

Since the Vehicle class doesn't have an explicit parent, it is automatically derived from Object.

As a result, all classes in the hierarchy also inherit the elements from Object as well as those from Vehicle.

Example: The High-Low Game

The user guesses a predetermined number in as few guesses as possible

The set up:

```
Scanner in = new Scanner(System.in);
Random generator = new Random();

int target = generator.nextInt(100) + 1;
int guess = -999;  // initial value out of range
int count = 0;

System.out.println("I've chosen a number " +
    "between 1 and 100.");
```

Example: The High-Low Game

```
while (guess != target)
{
    System.out.print("Guess what it is: ");
    guess = in.nextInt();

    count++;

    if (guess < target)
        System.out.println("Too low!");
    else if (guess > target)
        System.out.println("Too high!");
    else
        System.out.println("That's it! You got it in " +
            count + " guesses.");
}
```

Example: The High-Low Game

A sample run:

```
I've chosen a number between 1 and 100.
Guess what it is: 50
Too high!
Guess what it is: 30
Too low!
Guess what it is: 40
Too low!
Guess what it is: 45
Too low!
Guess what it is: 47
That's it! You got it in 5 guesses.
Thanks for playing.
```

EXCEPTION HANDLING

Exceptions

An exception occurs when something unexpected happens while a program is **running**.

An exception is an object that represents such an unusual or erroneous situation, such as:

- Attempting to divide by zero
- Attempting to follow a null reference
- An array index that is out of bounds
- A specified file could not be found
- Lots more!

Exceptions

Catching an exception enables a program to respond gracefully. For example, when the following line of code is executed:

```
double result = 12345 / 0;
```

An ignored or uncaught exception causes a message like:

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at DivideByZero.main(DivideByZero.java:11)
```

Exceptions

Java uses exception handling to enable a program to catch exceptions and respond to them programmatically.

The Java API has a predefined a set of exceptions that can occur during execution, and a programmer can create more.

There are 3 ways a program can handle an exception:

1. Ignore it (error message!)
2. Handle it where it occurs (try-catch statement)
3. Handle it somewhere else in the program (exception propagation)

The try-catch Statement

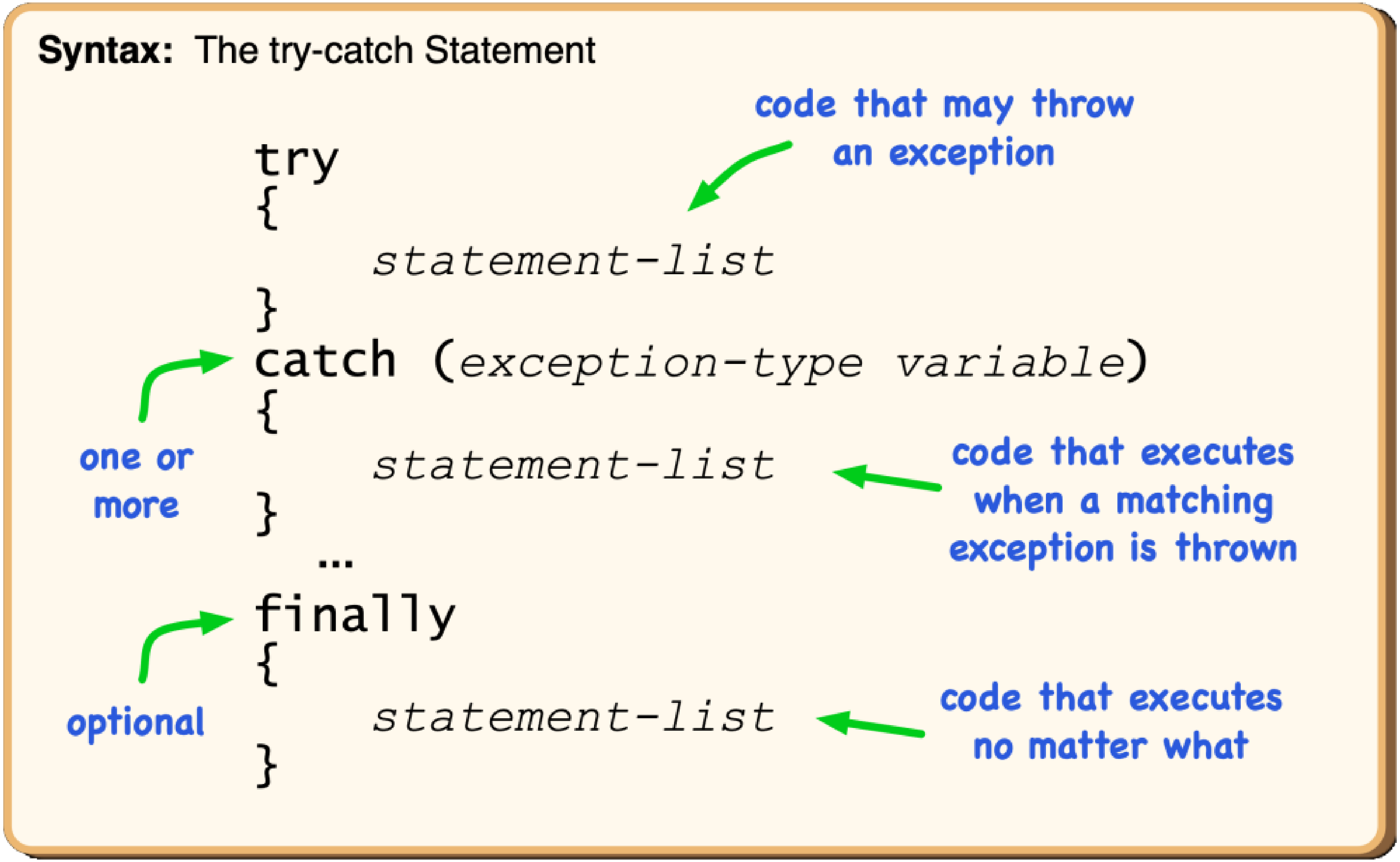
Catching an exception that is thrown due to a runtime error enables a program to respond rather than simply crashing.

In Java, this is done using the try-catch statement.

The 3 parts or blocks of a try-catch statement are:

1. try
2. catch
3. finally

The try-catch Statement



Syntax: The try-catch Statement

```
try
{
      statement-list
}
catch (exception-type variable)
{
      statement-list
}
   ...
finally
{
      statement-list
}
```

code that may throw an exception

one or more

code that executes when a matching exception is thrown

optional

code that executes no matter what

The try-catch Statement

This try-catch statement catches a NullPointerException.

```java
String myString = null;
try
{
    System.out.println("Length is: " + myString.length());
}
catch (NullPointerException e)
{
    System.out.println("Hey, that's a null reference!");
}
System.out.println("We're past the try-catch now.");
```

```
Hey, that's a null reference!
We're past the try-catch now.
```

The try-catch Statement

The parenthetical expression after the catch keyword is how the caught exception is made available to be handled and printed.

```java
try
{
    int result = 45 / 0;
}
catch (ArithmeticException e)
{
    System.out.println("Hey, don't divide by zero!");
    System.out.println("Message: " + e.getMessage());
}
```

```
Hey, don't divide by zero!
Message: / by zero
```

The try-catch Statement

An optional finally block can come after a try-catch statement. Code in the finally block is always executed, no matter what.

```java
try
{
    int result = 45 / 0;
}
catch (Exception e)
{
    System.out.println("Something horrible happened!");
}
finally
{
    System.out.println("But I'm ok with it now.");
}
```

```
Something horrible happened!
But I'm ok with it now.
```

Exception Propagation

If an exception is thrown but isn't caught in the method that threw it, it **propagates** up the call stack.

The exception may then be caught anywhere along the series of methods that were called to reach the point in the code where the exception occurred.

If the exception isn't caught anywhere in a try-catch statement, the program will crash… it will stop running with an error message describing the exception.

Exception Propagation

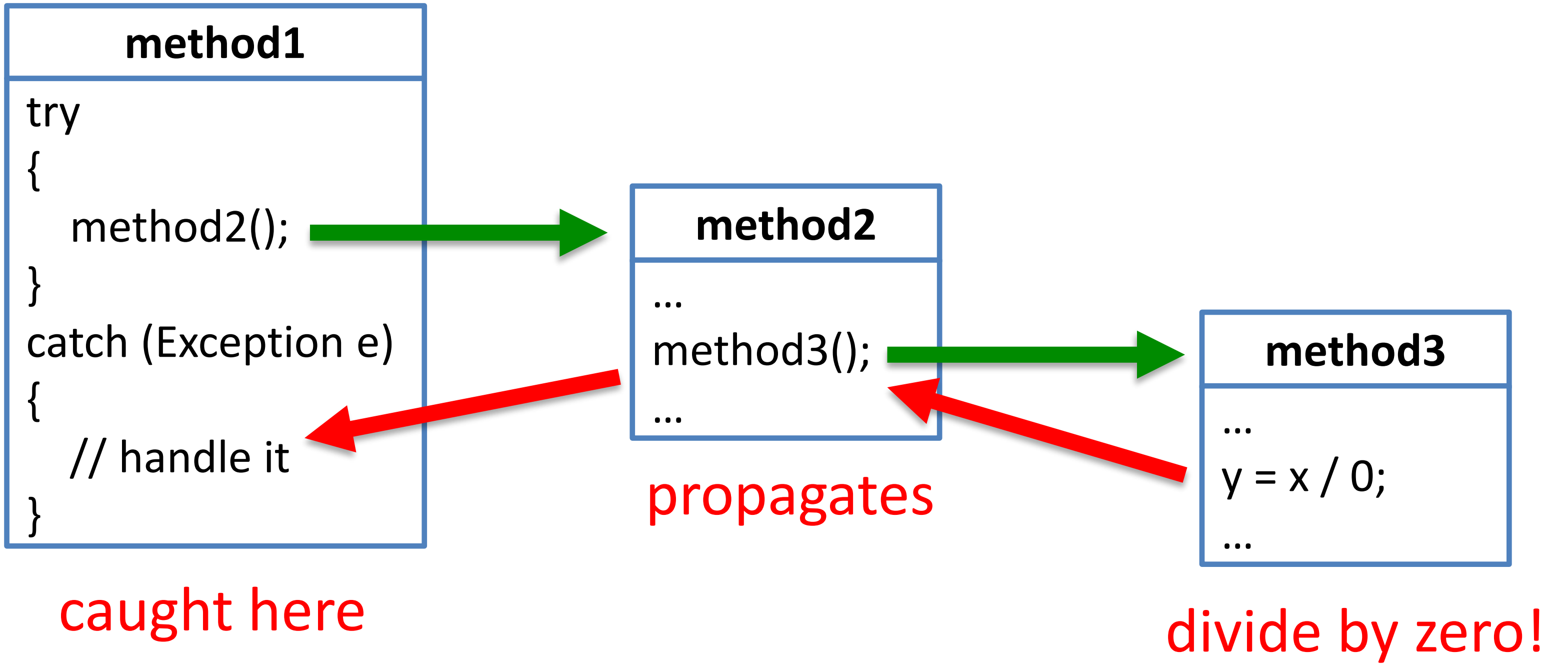
In this example, the exception throw by **method2** propagates to **method1**, where it is caught and handled.

```
public void method1()
{
    try
    {
        method2();
    }
    catch (Exception e)
    {
        System.out.println("Problem in method2!");
    }
}

public void method2()
{
    int nope = 5432 / 0;
}
```

Exception Propagation

A thrown exception affects flow of control in a way similar to a conditional statements (if and switch), repetition statements (for, for-each, and while), and method calls.



The throw Statement

For exception handling, the try-catch statement is how runtime errors can be dealt with programmatically in Java.

You may also want to write code to raise or **throw** your own exception. This is done using the throw statement, like this:

```
if (whoWasPwned.equals("me"))
    throw new Exception("I was pwned big time!");
System.out.println("The person pwned was " + whoWasPwned);
```

If the value of whoWasPwned is "me", the exception is thrown. Otherwise, the output looks like:

```
The person pwned was Weird Al
```

The throw Statement

Exceptions are defined by the java.lang.Exception class, or can be derived via inheritance.

When you construct a new Exception, the constructor accepts a custom message to associate with the it:

```
if (true)
    throw new Exception("Custom message goes here");
```

The throw Statement

Define your own exception class, derive it from Exception.

```
public class MyException extends Exception
{
    public MyException(String message)
    {
        super(message);
    }

    // Other methods can be defined here.
}
```

The constructor can call super to specify the message for the exception (see the Inheritance topic for details).

INTRODUCTION TO COLLECTIONS

Collections Overview

A collection is an object that manages a group of other objects.

The collection classes defined in the Java API are some of the most useful and versatile.

Each type of collection manages objects in a particular way:

- List – list of elements you an add, remove or replace
- Queue – list you can only add on one end & remove on other
- Stack – list where you add and remove only on one end
- Map – collection that uses keys to lookup values
- Set – unordered collection of unique elements

Collections Framework

Collections in Java are organized by a design architecture known as the Collections Framework.

The framework includes a set of Java interfaces that define the operations on a collection, as well as one or more classes that implement the interfaces.

Java Collections are defined in the `java.util` package.

Collections vs. Data Structures

These two terms sometimes get used interchangeably. For our purposes, they are defined as:

collection - manages a group of objects in a particular way

data structure - the programming technique used to implement a collection

The collection is the concept and the data structure is how it gets done.

Collections are Generic

All Java collection classes are **generic**.

A generic class is a class that specifies the type of data the class manages using a placeholder.

For example, the ArrayList class is named **`ArrayList<E>`**, where the E is a placeholder for the type of element to be stored. It is used like this:

```
ArrayList<String> nameList = new ArrayList<String>();
```

The for-each loop

Traversing a collection is made easy by the for-each statement, which is a variation of the for loop:

```java
for (Member mem : memberList)
{
    System.out.println(mem.getName());
    System.out.println(mem.getMembershipNumber());
}
```

On each iteration of the loop, the variable `mem` is assigned the next `Member` object from the list, starting with the first one. The loop repeats until the last object in the list is used.

PREVIEW OF STACKS

Stacks

A collection is an object that stores and manages other objects in particular ways

A stack is a collection whose elements are added and removed at one end (the top of the stack)

Think of a stack of books or a stack of boxes

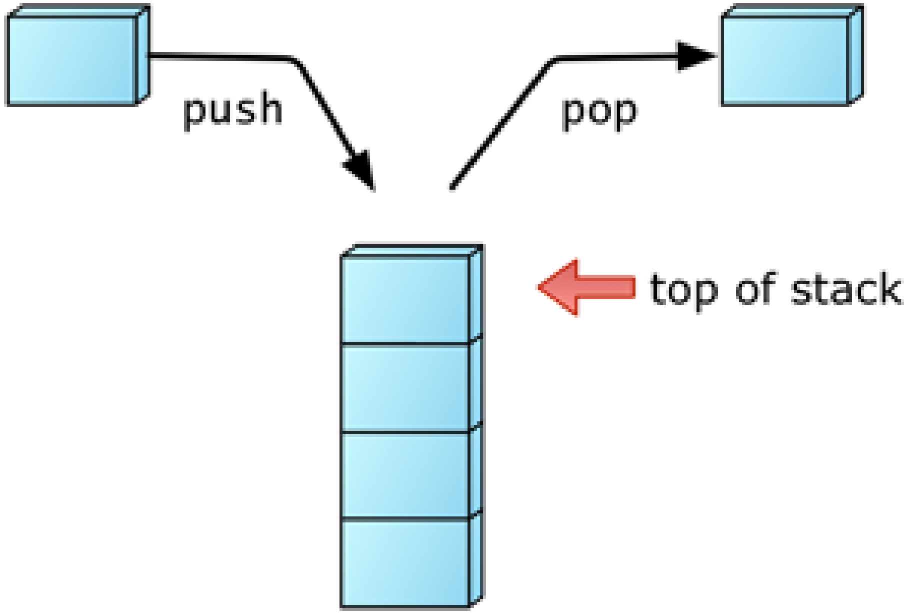
When an element is added, it becomes the new top of the stack

A stack can simplify the solution to many problems

Stacks

The operations on a stack have classic names

You push an item onto the stack and pop an item off



Stacks

You can also peek at the top item – examine and interact with the top item without removing it from the stack

You can also check whether a stack is empty (contains no elements)

A stack is a LIFO collection – Last In, First Out

The last element to be added to the stack is the first one to be removed

If you need to access other elements, you shouldn't use a stack

Another collection, such as a queue or a list, may be more appropriate