Java – a re-cap

introduction

Java is an object-oriented programming language

- Not *purely*, like Ruby, Smalltalk, Eiffel
  - More strictly than C++
  - But not everything is an object
- Everything belongs to a class
  - No global variables, nor functions
- Very portable
  - Programs run in a virtual machine (the *JVM*)

Java basics

- Built-in (primitive) types:

  `boolean byte char int long float double`
- literals
  - Primitives are *not* instances of a class
- Branches

  `if if-else switch ?:`
- Loops

  `for while do-while break continue`
- Comments
  - `/* ...*/` – block comment
  - `// ...` – line comment

Java basics (cont'd)

- `{}` – Defining and nesting scopes
- `;` separates statements
  - Not used to end class definitions
- Exceptions
  - `throws( errorlist )`
  - `throw error`
  - `try{}`
  - `catch{}`
  - `finally{}`
- No user-overloading of operators
  - No I/O operators, « »
  - `+ -` etc. don't work with `Integer`, `Float`
- No multiple inheritance
- No explicit pointer notation

Some features of java

- Interfaces
  - Rather than multiple inheritance
- Iterating `for` loop

  `for( declaration : iterable )`
- Java won't accept `int`s, nor `elephant`s, where a `boolean` is expected
- All methods are virtual
- Inner classes
- `/** ...*/` – Javadoc comment

Other features of java

- No global anything
  - Everything is inside a class
- Single inheritance tree
  - `Object` is common ancestor of all classes
- Variables of type Object are references
  - Reference semantics apply, for assignment, passing into / out of functions
- Generics
  - Type-safe containers
  - Cannot hold primitive types
    - All primitive types have wrapper classes
    - Boxing and unboxing

reference vs copy

Semantics – objects vs primitives

- Variables of any primitive type use copy semantics, as in C
- Assignment and equality work as expected:
  - Assign the value stored in `b` to `a`

    ```
    a = b
    ```
  - Compare values stored in 2 different variables

    ```
    a == b
    ```

references

- If a variable is of type `Object` (an instance of some class that inherits from `Object`), it is a *reference*
  - Stores the object's location in memory
  - A pointer, without the notation
- Assignment and equality have different meaning:
  - Assign two references to the same object

    ```
    a = b
    ```
  - Check if both references refer to the same object

    ```
    a == b
    ```

references (cont'd)

- Use `clone` method to create a copy of an object
- Use `equals` method to see if two distinct objects are equivalent
  - Inherited from `Object`
  - Should be overloaded if object, in turn, contains references to other objects
- Deep vs shallow copy

strings

`String` class

- The only built-in non-primitive
  - Compiler knows about it
  - Compiler wraps string literals in a `String` object
- `String`s are *immutable*
- Concatenation
  - + works with `String`s
  - Primitives are *coerced* into `String`s

arrays

Primitive arrays

- Much like C arrays
  - Allocated from the heap:

```
int [] ia = new int[ 20 ] ;
int [] ja = { 12, 8, 392 };
```

  - Use reference semantics
- A single, final attribute, `length`
- Indexed using `[]`

Array example


```
import java.io.*;

public class Example {
  public static void main( String [] argv ) {
     int sum = 0;
     int [] temps = { 65, 87, 72, 75 };
     for( int i=0; i<temps.length; ++i )
        sum += temps[i];
     System.out.print( "# of samples: " + temps.length );
     System.out.println( ", avg: " + sum/temps.length );
  }

}
```

Resizing an array


```
public class ArrEg {
  public static int[] resize( int [] a, int newsize ) {
    int [] rv = new int[newsize] ;

    for( int i=0; i<a.length; ++i )
      rv[i] = a[i] ;
    return rv ;
  }

  public static void main( String[] args ) throws Exception {
    int [] a = { 74, 011, 23, 0xff };
      // Want to add more items. Get bigger array
    int [] t = resize( a, 2*a.length ) ;
    a = t; // the old array is now marked for deletion
    t = null;
    a[4] = 47 ;
    for( int i : a )
      System.out.print( i + ", " ) ;
  }
}
```

Iterating `for` loop

- Works with primitive arrays, and any generic, `Iterable` container

```
int [] temps = { 65, 87, 72, 75 } ;
for( int i : temps )
   sum += i ;
```

- An example of boxing and unboxing

```
import java.util.ArrayList ;

public class Al {
  static public void main( String [] args ) {
    ArrayList<Integer> v = new ArrayList<Integer>( ) ;
    v.add( 72 ) ;
    ...
    for( int i : v )
      ... ;
  }
}
```

classes

Java classes

- Each class goes into a separate file
  - Class `Foo` would be in a file named `Foo.java`
  - To compile:

```
$ javac Foo.java
```

- Each class can have a static `main` method
  - To run, tell the JVM which class to start in (who's `main`):

```
$ java Foo
```

Java classes (cont'd)

- Each class needs an access modifier
  - Default is package
- Each member takes an access modifier
  - Default is package

Default Packages

Without an implicit package specifier, all classes in the same directory are in the same package.

- All methods are virtual
- Static attributes can be initialized at declaration

`final` modifier

- Attributes:
  - Must be initialized
  - If a primitive type, value cannot be changed
  - If a reference to an `Object`, the *reference* may not be changed
    - But the object referenced may still be modified
- `final` methods may not be overridden
- `final` classes may not be extended

Static attributes

- Also called a *class attribute*
- A single variable, shared by all instances of the class
  - Don't need an instance to access
- Consider the interest rate on `SavingsAcct` class
  - Each instance would have its own account number, etc.
  - Each would *share* today's interest rate:

    `SavingsAcct.rate`
- `static public` is how we implement system-wide globals in Java
  - Consider `Math.Pi`

Static methods

- Cannot access instance data
- Don't need an instance to access

```
SavingsAcct.getRate() ;
```

- `static public` is how we implement "global" library functions
  - Consider `Math.sin()`, `Math.log()`, `Math.floor()`, etc.

Access modifiers

`private`	No access outside class
default	package – only to classes in package
`protected`	Access given to classes in package, subclasses
`public`	All have access

- A class may be either *public*, or default
- Class members may have any modifier
- Note, member modifiers can *not* grant accesses not granted by class

`main` method

- Every class may have a `main`

```
public static void main( String [] args )
```

- Entry point, potentially
  - Class to start in must be identified to JRE
  - No instances yet, so, must be `static`
- No return value from `main`
- Use static methods for a traditional C-like program

Command-line arguments

- Single parameter to `main`
- Java array of `Strings`
- Argument 0 is *not* the name of the program, class, etc.

```
public static void main( String [] args ) {
   for( int i=0; i<args.length; ++i )
      System.out.printf( "%d %s\n", i, args[i] ) ;
}
```

Constructors

- No return type
- If none is provided, default constructor is used
- If *constructor* is provided, default is no longer implicitly available
- There is no destructor
  - See `close()` and `finalize()`
    - `finalize()` is unreliable
    - Should not be used as a destructor
    - Output streams should be closed explicitly

super

- Use it to call one of the parent class' constructors, to initialize the parent sub-object

  `super( name ) ;`

  - Place it as first line in child's constructor
  - If absent, parent's default constructor is called
- Use it to call parent's version of overridden method

```
public class Professor extends Person {
  public String toString() {
    return "Prof. " + super.toString() ;
  }
  ...
}
```

Importing classes from packages

- You can import individual classes:

```
import java.io.ObjectInputStream ;
```

- Import all classes in a particular package
  - E.g., all classes in the `java.net` package:

```
import java.net.* ;
```

Classes as namespaces

The `Math` class, e.g., is simply a container for methods and constants.

- Can't be instantiated
  - The default constructor is made `private`
- The class is `final` – can't be subclassed
- All methods are `public static`
  - `Math.sin( a ) ;`
  - `Math.exp( x ) ;`
- Constants are `public static` attributes

i/o

Input / Output

- Many classes
- Note the difference between reading ASCII input vs raw data
- Choose the right one for the job
- Always call `close` explicitly
  - Especially output streams
  - No guarantee that `finalize` will be called

Text input – Scanner

Very handy class for formatted ASCII input

- Can be wrapped around:
  - File

```
Scanner src = new Scanner( new FileReader( "data" )) ;
```

  - InputStream

```
Scanner src = new Scanner( System.in ) ;
```

- Scanner will open a file for you

```
Scanner src = new Scanner( new Path( "../Files/input.src" )) ;
```

Scanner examples

- Can read by lines:

```
while( src.hasNextLine() )
   l = src.nextLine() ;
```

- By words:

```
String s ;
while( src.hasNext() )
   s = src.next() ;
```

- Or, by tokens, over the primitive types

```
int i ;
while( src.hasNextInt() )
   i = src.nextInt() ;
```

Use `Scanner` to parse a `String`

- By default, token delimiter is white space
  - Can be changed
  - Can use a regular expression (`Pattern`) to describe the delimiter

```
String s = "Parse-*-this-*-up" ;
Scanner src = new Scanner( s ).useDelimiter( "-*-" ) ;
ArrayList<String> fields = new ArrayList<String>() ;

while( src.hasNext() )
   fields.add( src.getNext() ) ;
```

Use `PrintStream` to write text

- `System.out` and `System.err` are instances of `PrintStream`
- `print` , `println` , `printf` overloaded for all primitives
- Can be wrapped around a `File` or an `OutputStream`
- Will open a file, given a `String`

```
PrintStream f ;
if( argv.length == 0 )
   f = System.out ;
else
   f = new PrintStream( argv[0] ) ;
...
f.close() ;
```

Reading raw data

Use `BufferedInputStream` or `FileInputStream` to read Bytes.

```
import java.io.BufferedInputStream ;
import java.io.IOException ;
public class readBytes {
  public static int wordLen = 8 ; // # of bytes to be read each time

  public static void main( String [] argv ) throws IOException {
    BufferedInputStream is = new BufferedInputStream( System.in ) ;
    byte [] buff = new byte[wordLen] ;
    int r ;
    while( (r=is.read(buff, 0, wordLen)) != -1 ) {
      for( int i=0; i<r; ++i )
        System.out.printf( "%x ", buff[i] ) ;
      System.out.print( "\n" ) ;
    }
    is.close() ;
  }
}
```

Writing raw data – `PrintWriter`

- Use `PrintWriter` to write raw data
- `print`, `println`, `printf` provide familiar behavior
- There are a handful of `write` methods, for writing characters and strings

inheritance

Inheritance vs aggregation

- Inheritance is the *is-a* relationship
  - A square is a shape
  - An employee is a person
  - A professor is an employee
  - So, a professor is a person
- Aggregation is the *has-a* relationship
  - A square has a color
  - An employee has an address
  - A car has an engine
    - And 4 tires

Inheritance

Given a simple class:

```
public class Person {
   protected String _name ;
   public Person( String n ) { _name = n ; }
   public String getName() { return _name ; }
}
```

We can define a subclass:

```
public class Professor extends Person {
   protected String _id ;
   public Person( String n, string i )
   {  super(n) ; _id = i ; }
   public String getName()
   { return "Prof. " + _name ; }
}
```

Abstract superclass

- An abstract class cannot be instantiated
- It typically contains method declarations, without definitions
  - These are behaviors that subclasses must provide to be meaningful objects
  - Use the `@override` annotation
- E.g., a closed shape might well know its color, and declare a method to compute its area
  - All closed shapes have an area
  - Computed differently for each shape
  - Area of an abstract shape is meaningless

Abstract superclass – example


```
public abstract class ClosedShape {
   protected Color _fill ;
   public ClosedShape( Color c ) { _fill = c ; }
   public Color color() { return _fill ; }
   public abstract double getArea() ;
}
```

We create an actual shape

```
public class Circle extends ClosedShape {
   protected double _radius ;

   public Circle( Color c, double r )
   {  super(c) ; _radius = r ; }

   @Override
   public double getArea()
   { return Math.Pi * _radius * _radius ; }

   public double getRadius() { return _radius ; }
}
```

Casting

- Objects can be cast up the tree
  - Always safe
  - Explicit cast not needed
  - Only methods from ancestor may be called

```
ClosedShape s = new Circle( Color.BLUE, 3 ) ;
```

- Objects can be cast down the tree
  - Might throw `ClassCastException`
  - Only methods from ancestor may be called

```
Circle c = (Circle) s ; // This works fine
Square q = (Circle) s ; // This throws exception
```

`instanceof` operator

- We can test objects
- An object of a subclass is always an instance of an ancestor
- An object of a class is not, generally, an instance of a descendant

```
Circle c = new Circle( Color.PURPLE, 8 ) ;
...
if( c instanceof Square ) System.out.println( "c is a Square" ) ;
if( c instanceof Circle ) System.out.println( "c is a Circle" ) ;
if( c instanceof ClosedShape )
   System.out.println( "c is a ClosedShape" ) ;
```

```
c is a Circle
c is a ClosedShape
```

Containers of shapes

We can create containers of shapes:

```
public static void main( String [] args ){
   ArrayList<Shape> shapes = new ArrayList<Shape>() ;
   shapes.add( new Circle( 3, 1 )) ;
   shapes.add( new Square( 5, 1 )) ;

   for( Shape s : shapes ){
      System.out.printf( "Area: %.2f\n", s.getArea() ) ;
   }
}
```

```
Area: 28.27
Area: 25.00
```

interfaces

Interfaces

- Java does not support multiple inheritance
  - This is not a bad thing
  - Multiple inheritance is messy, both in design and implementation
- An interface describes behaviors which must be supplied by any implementing class
  - It *declares* methods
  - It does not *define* any
  - Attributes, however, *can* be defined

Interfaces (cont'd)

- An interface can "inherit" from one or more other interfaces:

```
public interface Stealthy {
   public void stalk() ;
}

public interface Predator extends Stealthy {
   public void pounce() ;
}
```

- A class might implement multiple interfaces:

```
public class Cat extends Animal implements Predator
{ ... }
```

exceptions

Library exceptions

- These are the exceptions in the Java standard library
- Exceptions in darker boxes are *checked* exceptions
  - Must be caught, or listed in a `throws` statement
- Inherit from any of these to make your own exceptions
  - No behavior need be defined
  - Its value is its type

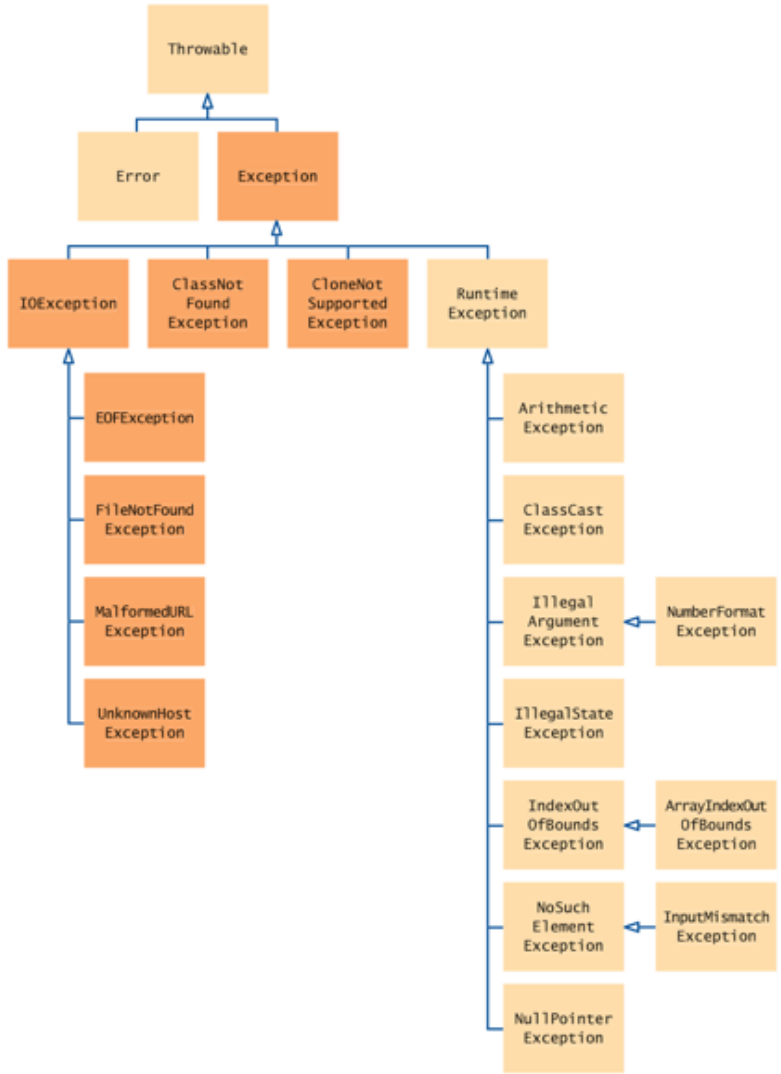

Figure: From C. Horstmann's Big Java, 4th ed.

User-defined exceptions


```
public class ThatsOdd extends IllegalArgumentException {
   public ThatsOdd( String s ) { super( s ); }
}
```

Can be used as any other exception:

```
public static void foo( int i ) throws ThatsOdd {
  if( i%2==1 )
    throw new ThatsOdd( "We're partial to evens, in this method." ) ;
  ...
}
public static void bar( int n ) {
  try {
    foo( n/2 ) ;
    ...
  } // try
  catch( ThatsOdd e ) {
    System.err.printf( "bar> caught ThatsOdd: %s\n", e.toString() ) ;
    e.printStackTrace() ;
  }
}
```

nested classes

Nested classes in Java

- Java allows classes to be defined inside other classes
  - Even inside methods
  - Even unnamed
- We'll briefly look at these, describe common uses

Types of nested classes

- The possibilities are:
  - Static nested class
  - Non-static inner class
    - Need an instance of outer class to instantiate
  - Defined inside a method (*method-local*)
    - Can only be instantiated in that method
    - Object can be returned from method

Public static nested classes

- Just a container for similar classes

```
public class Public {
   public static class Inner1 {
      public void talk() {
        System.out.println( "In Public.Inner1.talk" ) ;
      }
   }

   public static class Inner2 {
      public void talk() {
        System.out.println( "In Public.Inner2.talk" ) ; }
   }
}
```

Public static nested classes (cont'd)

- Used like any other class
- Note the scope

```
public static void main( String [] args ) {
   Public.Inner1 i1 = new Public.Inner1() ;
   Public.Inner2 i2 = new Public.Inner2() ;

   i1.talk() ;
   i2.talk() ;
}
```

Non-static nested classes

- Also called *inner* classes
- Needs an instance of outer class
- These objects capture their surrounding scope, even if the containing object is no longer accessible
- Can be an alternative to exposing outer class' attributes to entire package (or world)
- An inner class may be *unnamed*
  - Commonly used to install event handlers

Non-static nested classes (cont'd)

E.g., inside a Dialog we might have event handlers for some controls

```
buttonYes = new JButton() ;
buttonNo = new JButton() ;
...
buttonYes.addActionListener(
  new java.awt.ActionListener() { // class definition here
    public void actionPerformed( java.awt.event.ActionEvent e ) {
      doSomething();
    }
  }
)
```

Factory methods

- Object creation is abstracted
- Class will choose which subclass to create
- Subclasses are hidden (can't be instantiated directly)
  - Private static nested class, or
  - Method local (defined inside the factory method)
- Consider the Sorting Hat, from Hogwarts
  - It decides which House a student belongs to
  - All houses have same interface:

```
public abstract class House {
   ...
   public abstract represent() ;
}
```

Factory methods – Hogwarts

- Inside we could define our subclasses:

```
private static class Gryffyndor extends House {
  @Override
  public void represent() {
      System.out.println( "Gryffyndor!" );
  }
}

private static class Slytherin extends House {
  @Override
  public void represent() {
      System.out.println( "Slytherin!" );
  }
}

private static class RavenClaw extends House {
  @Override
  public void represent() {
      System.out.println( "RavenClaw!" ) ;
  }
}
```

Hogwarts (cont'd)

- Here's our factory method, with exclusive access to the subclasses:

```
public static House SortingHat( int i ) {
  House rv = null ;
  switch( i%3 ) {
    case 0 : rv = new Gryffyndor() ; break ;
    case 1 : rv = new Slytherin() ; break ;
    case 2 : rv = new RavenClaw() ; break ;
  }
  return rv ;
}
```

- We let the factory decide the proper subclass to use:

```
public static void main( String [] args ){
  House h = House.SortingHat( 27 ) ;
  h.represent() ;
}
```

containers

Standard library containers

- Some standard, useful containers
- These are *generic* containers
- Can only hold `Objects`, and descendants (no primitives)
- Many (but not all) implement the `Collection` interface
  - The others probably implement `Iterable`

Generics

- Allow containers to hold *particular* `Objects`
  - Makes code more type-safe
- Primitives are automatically *boxed* into appropriate objects (`Integer`, `Double`, etc.) when inserted
  - And unboxed when returned

The `Iterable<T>` interface

- Any implementing class has these methods:
  - `iterator` – Returns an `Iterator` over the elements
  - `spliterator` – Returns a `Spliterator` over the elements
  - `forEach( Consumer<? super T> action)` – Applies `action` to each element of the Iterable

The `Collection<T>` interface

- Inherited from the Iterable Interface
- Also provides the following (among others):
  - `add`, `addAll`
  - `contains`, `isEmpty`, `size`, `clear`
  - `remove`, `removeIf`, `retainAll`

The `Collections` algorithms

- Many handy algorithms that work on objects that implement `Collection<T>`
- These containers apparently have some notion of indexing
  - (This doesn't imply constant-time access)
- Here are a few handy ones:
  - `max`, `min`, `binarySearch`, `sort`, `reverse`, `shuffle`, `sort`
  - `replaceAll`, `swap`, `fill`, `frequency`

ArrayList<T> – a Vector

- The preferred vector, these days
- Inherits from `AbstractList`
- Implemented interfaces include `List<E>`, `Collection<E>`, `Iterable<E>`
- Some useful methods:
  - `add( T elem[, int index] )`
  - `clear()`
  - `contains( Object elem )`
  - `get( get( int index )`
  - `set( int index, T elem )`
  - `size()`
  - `iterator()`

LinkedList<T>

A doubly-linked list implementation of the `List` interface

- No constant-time access of elements
- Can modify anywhere in constant time
- Interfaces include `Collection<E>`, `Iterable<E>`, `Deque<E>`, `List<E>`, `Queue<E>`
  - Note, there is no stack interface
  - `Stack<E>` is a class, built on `Vector<E>`

ArrayDeque<T>

- Deque – doubly-ended queue
- Supports constant-time insert and delete at front, also
- Much like a vector
  - Indexed (constant-time access)
  - Modification of middle still linear operation
- Interfaces include `Collection<E>`, `Deque<E>` and `Queue<E>`

The `Set<T>` interface

- Implemented by:
  - `HashSet<E>`, `LinkedHashSet<E>`, `TreeSet<E>`
- Behaviors include:
  - `contains`, `add`, `remove`

The `Map<K,V>` Interface

- Implemented by several classes, including:
  - `HashMap<E>`, `LinkedHashMap<E>`, `TreeMap<E>`
- `HashTable` is a child of `Dictionary`
- Behaviors include:
  - `clear`, `hasKey`, `hasValue`, `get`, `remove`

jar

Java ARchives

- Use `jar` utility to create jar files
- A *jar file* is:
  - A compressed, tar'd collection of java files
  - Uses the PKZIP library
- Jarfile typically contains files necessary for a program, system, applet, etc.
  - `.class` files
  - Any other resources (images, sound files, etc.)
  - META/MANIFEST.MF – Jar metadata
- Jarfiles natively understood by JVM
  - Can be added to classpath
  - Searched for classes, like a directory

Invoking `jar`

- Options much like tar
  - `c` – create
  - `x` – extract
  - `u` – update
- Others
  - `v` – verbose
  - `f` *file* – specify archive filename
  - `m` *file* – include manifest information from *file*

Creating a jar file – example

- Given the following files:

  `pokerSuit.class pokerClient.class cheesy.mp3`
- Create the jar file:

```
$ jar cvf poker.jar poker*.class cheesy.mp3
```

- Let's take a peek at the file
  - Note the manifest file

```
$ unzip -l poker.jar
Archive: poker.jar
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2016-07-21 17:38   META-INF/
       69  2016-07-21 17:38   META-INF/MANIFEST.MF
      868  2016-07-21 17:38   pokerClient.class
     1513  2016-07-21 17:38   pokerRank.class
     1102  2016-07-21 17:38   pokerSuit.class
        0  2016-07-21 17:37   cheesy.mp3
---------                     -------
     3552                     6 files
```

Executing a jar file in the jvm

A Jarfile can be executed by the JVM:

- Identify the main class in a manifest file

poker.mf

```
Classpath: ./poker.jar
Main-Class: PokerClient
```

- Create the jar file with this information:

```
$ jar cvfm poker.jar poker.mf poker*.class cheesy.mp3
```

- To run:

```
$ java -jar poker.jar
```

credits

thanks

The contents of these slides were created by Kurt Schmidt and modified by other faculty of the Drexel University CS Department including Geoffrey Mainland, Bruce Char, Vera Zaychick, Jeremy Johnson, Spiros Mancoridis, and others.