- review Java references
- draw pictures to design data structures
- simple dynamic implementation
- use an interface
- make generic
- add exceptions

Next week and homework
- next week: stack applications
- homework by next week:
  – download, run and understand this week's example programs
  – read Horstmann, section 15.6 on stack applications

Lab
- exercise assigned this week (is not collected or graded) / reviewed next week

Review last week

- introduced stack characteristics and primitive operations in abstract:

```
s.push(i);
i = s.pop();
i = s.top();
s.empty();
```

- stacks are commonly used to reverse something, or to backtrack
- looked at several example applications
- did simple array implementation of stack as homework

Introduction to this week

- this week, want to develop a dynamic implementation of stack, using a linked list
- start with a review of Java references
- then emphasize an important technique where we draw pictures to design complex data structures
- complete first a simple implementation of stack
- then add more advanced Java features to it
- will practice using our final dynamic implementation in this week's programming exercise

Review Java references

Objective: a reference in Java is like a pointer in C, but simpler and safer

The job of a reference is to ‘point to’ or ‘refer to’ an object

- it is best to think of a Java reference as “a variable with a name, its job is to ‘point to’ (or ‘refer to’) an object in memory”
  - is most clearly shown as a picture e.g. here’s a reference named r, referring to an object in memory:

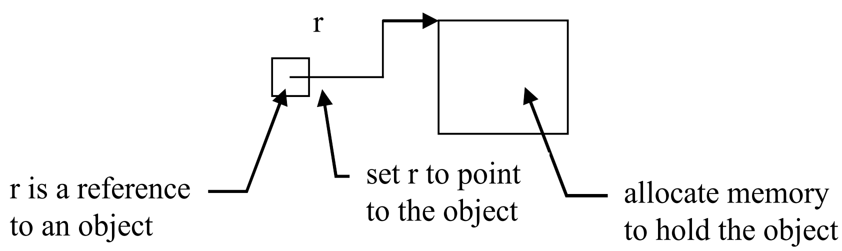

Figure 1 a reference r referring to some object

  - (actual implementation of a reference in Java may be more obscure than this. Just keep this simple picture in mind)

A reference is like a pointer in C, but safer

- we can do only a limited number of things with a Java reference:
  - a reference can refer only to objects of the appropriate data type
  - we access the object via the reference: its constants, instance variables and methods, controlled by private, protected, public access
  - a reference can be changed to refer to a different object of the appropriate data type
  - and that’s it!
  - very much more limited, simpler and safer than using pointers in C!

Several references can refer to the same object

- this is a normal and important part of data structures programming. Here's an example using the Java library `Rectangle` class, showing the pictures of main memory:

```
Rectangle r1 = new Rectangle(10, 20);
```

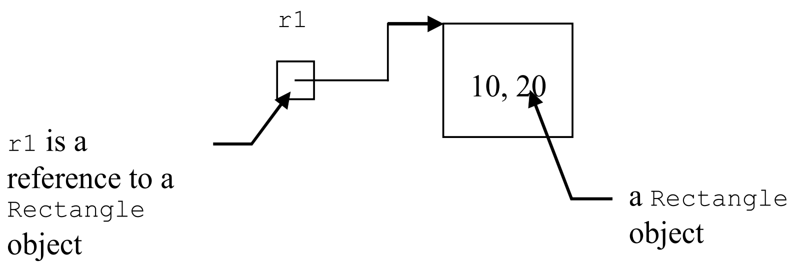

Figure 2 reference r1 refers to a new Rectangle object

- the essential idea is that assignment to a reference sets which object it refers to. The assignment operator '`=`' with references can be read as "becomes a reference to the same object as"

  – e.g., continuing from the previous line of code:

```
Rectangle r2 = r1;
```

  – means `r2` becomes a reference to the same object as `r1`, gives:

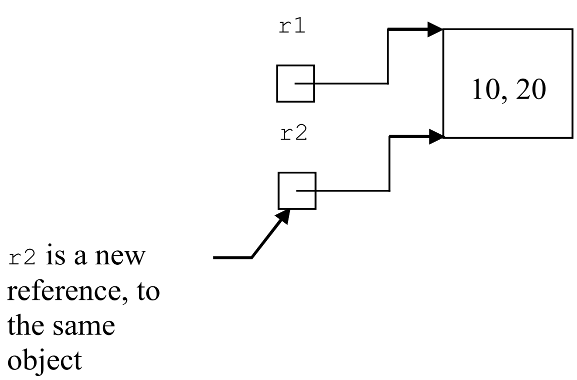

Figure 3 r2 becomes another reference to the same Rectangle object

  – see this? – the assignment to the new reference above means that `r2` becomes a reference to the same object that `r1` refers to, as shown

- demonstrate this effect by changing `r2` then printing `r1`:

```
r2.setSize(30, 40);
System.out.println(r1);
```

  – look at the picture, follow the two references, what do you think is printed for `r1`?

  – run the example program to check this…

- from Canvas, 'Dynamic implementation of stack' module, Example programs, download, read, run and understand my `Review references` example program

  – understand here how the assignment to a reference `r2 = r1` sets which object `r2` refers to

Automatic garbage collection is safer, and prevents memory leaks

- an 'orphan object' is "an object in memory that no references point to"

  – e.g. start from this simple situation:

```
Rectangle r1 = new Rectangle(10, 20);
Rectangle r2 = new Rectangle(30, 40);
```

  – in memory:

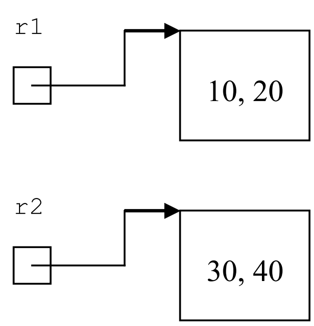

Figure 4 two separate Rectangle objects

  – two separate `Rectangle` objects

- now deliberately create an orphan:

```
r2 = r1;    // deliberately create an orphan object
```

  – this means `r2` becomes a reference to the same object as `r1`, gives:

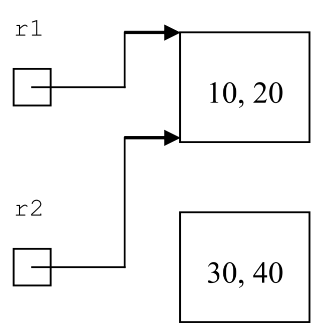

Figure 5 the 30, 40 rectangle has become an orphan

- see that no references point to the (30, 40) rectangle, so it has become an orphan
  - effectively this object no longer has a name. We can no longer access it
  - so it should be disposed of
- Java's 'garbage collection' strategy automatically finds orphans and returns the memory to the operating system memory manager for use elsewhere
  - safe but expensive, as we'd expect of Java
- this automatic memory management prevents 'memory leaks', where bad programmers continually acquire memory but never free it
  - a significant programming error
- (note that C-style manual memory management is so difficult to do correctly, and so prone to programming errors, that the C memory management operators do not even exist in Java:
  - `&` – 'address of' operator
  - `*` – 'dereference' operator
  - `malloc()` – allocate memory
  - `free()` – de-allocate memory
  - 'pointer arithmetic' – can process arrays in C using pointers not indexes)

<u>Everything in Java is a reference!</u>

- in the beginning with Java, everything looks like an object e.g.

```
Integer r = 9;
```

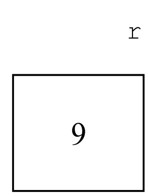

Figure 6 an Integer object named r

- we think of `r` as an `Integer` object. So we know to call `Integer` methods for this object

- this is a normal and productive viewpoint, works fine

• however, with more experience, we realize that everything in Java is actually a reference to an object

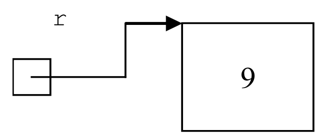

Figure 7 reference r refers to a new Integer object containing value 9

- this is a deeper and more accurate understanding

- gives us more power as programmers, that now we can also manipulate the references, to change which object they refer to…

- this realization is the beginning of programming for data structures!

<u>Summary</u>

• Java references are similar to pointers in C, but are more restricted and safer. Exactly as you would expect

- a lot of the problems with pointers in C are impossible with Java references, since there is strict type checking, and no `&` and `*` operators, pointer arithmetic

- also, Java's automatic garbage collection of orphan objects is safer than manual memory management, and prevents memory leaks

Draw pictures to design data structures
Objective: review an essential technique that you probably know already

How to design data structures

- drawing pictures is an essential technique when designing complex algorithms, for dynamic data structures or for any other significant problems:
  - draw picture of a start state
  - identify all possible cases
  - use pictures to design algorithms that get to required end state
  - then write algorithms in pseudocode or Java

Summary

- drawing pictures makes designing complex code easier
  - you focus on what to do and how to do it
  - ignoring trivial syntax
- only draw pictures if it's useful!
  - is an essential technique for complex data structures algorithms, to explore a way to a solution and keep track of everything
  - then with more experience you can often design algorithms directly iu Java
- many examples of how to do it coming next…

Simple dynamic implementation of a stack
Objective: see how to draw pictures to design and implement a simple stack dynamically, using references

Will use references to implement a stack dynamically:

- draw the picture e.g.

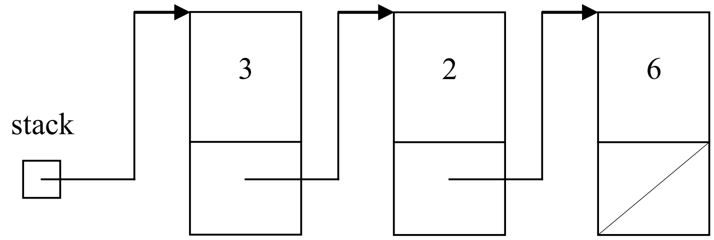

Figure 8 a stack implemented dynamically

  – declare a reference named `stack`
  – points to the top item of the stack
  – `null` for an empty stack
  – add and remove items at the front of the list as we push and pop the stack

The stack contains items

- want to make the items in our stack simple. Here's the `Item` class:

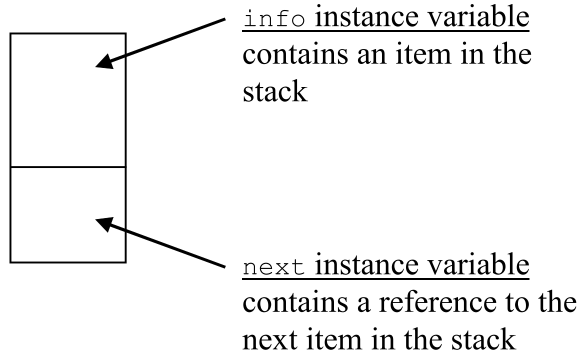

Figure 8 the simple `Item` class

  – `info` is just an `int`, for simplicity
  – `next` is the reference to the next item in the stack

- will declare these as protected for our convenience, so they can be directly accessed by classes that use the `Item` class
  - (otherwise we would write get and set methods to access instance variables declared private)

Implementation of the `Item` class

- here's how to declare a reference variable in Java:

```
public class Item
{
    protected int info;
    protected Item next;
    . . .
```

  - see that `next` here is a reference to an `Item`. Exactly what we want!
- now begin to show how to use references by implementing the `Item` methods
  - keep it as simple as possible
- the constructor creates a new empty item:

```
    public Item()
    {
        info = null;
        next = null;
    }
```

  - `null` is a java keyword, for the null reference or pointer
- we need to create a new `Item` object when we push a new integer to the stack. So an overloaded constructor that take a parameter:

```
public Item(int i)
{
    info = i;
    next = null;
}
```

- this gives us the `Item` class we need for this first simple example

Implementation of the `Stack` class

- the `Stack` class maintains a reference `top` to the top item on the stack:

```
public class Stack
{
    private Item top;
```

- to create an empty stack simply set this to null:

```
    public Stack()
    {
        top = null;
    }
```

- is the stack empty?

```
    public boolean isEmpty()
    {
        return top == null;
    }
```

<u>`push()`. Draw pictures to design methods!</u>

- e.g. push an item 7 to the top of a stack

- is best to design methods by first drawing pictures then writing pseudocode or Java

  – drawing helps us see that there are 2 situations to handle here – when stack is empty or not. Example start states:

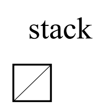

Figure 9 case when stack is empty

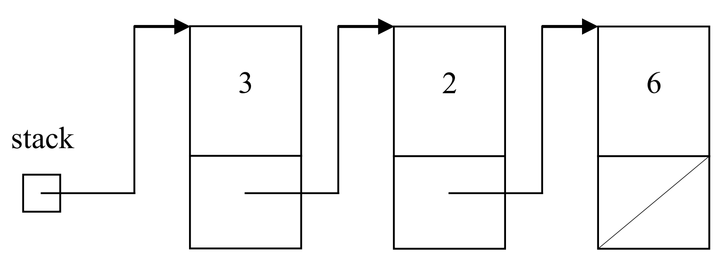

Figure 10 case when stack is not empty

  – now play with the drawings to figure out what steps will get us to the required end states, pushing the new 7 item to the top of the stack:

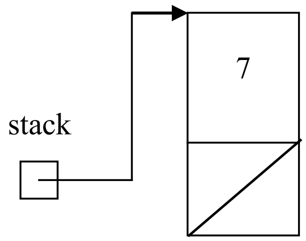

Figure 11 required end state for the empty stack

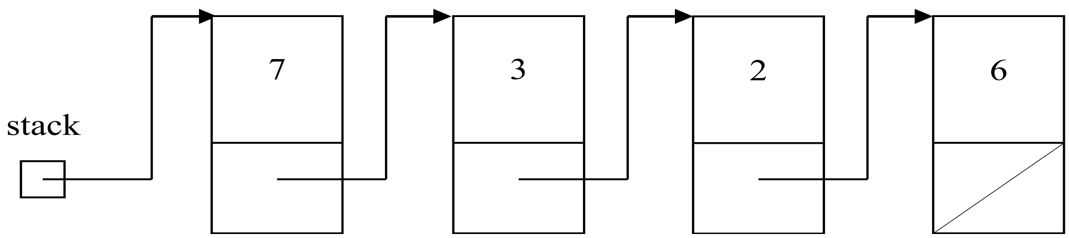

Figure 12 required end state for the not-empty stack

- once you've designed a good solution with pictures, then turn it into pseudocode and/or Java. So something like

  – in pseudocode:

  create new Item
  if stack is not empty
      set item's next to the top of the stack
  set top to new item

  – in Java:

```
public void push(int i)
{
    Item item = new Item(i);

    if (!isEmpty())
        item.next = top;

    top = item;
}
```

  – BTW, avoid implementation dependent code where possible e.g. use `isEmpty()` here rather than `if (top != null)`

<u>pop()</u>

- pop removes the item from the top of the stack

- pictures show 2 cases again – stack is empty or not. Here's a !empty stack:

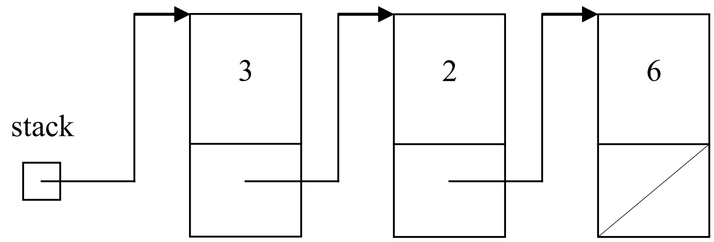

Figure 13 example of a !empty stack

- to simplify, for now will assume that pop is called only when the stack is not empty. Will handle underflow correctly soon

- after drawing, then the code. Here's pseudocode:

  if stack is empty
     report underflow error
  read item from top of stack
  set top to point to next item

  - then Java:

```
public int pop()
{
    //assumes stack is not empty
    int temp = top.info;
    top = top.next;
    return temp;
}
```

Test the new classes

- from Canvas, 'Dynamic implementation of stack' module, Example programs, download and unzip my `Simple stack` example program now

- read through `Tester::main()`, see how the program pushes a bunch of ints to a stack then pops them all off:

```
public static void main()
{
    Stack s = new Stack();

    for (int x = 1; x < 6; ++x)
        s.push(x);

    while (!s.isEmpty())
        System.out.println(s.pop());
```

```
        System.out.println("done");
    }
```

- run the program, see that it tests this first, simple, dynamic implementation of stack

Summary

- we've seen the Java syntax to implement references or pointers, and implemented a first, simple, dynamic stack
- emphasized here the important technique that drawing pictures is the best way to design data structures!
  - so that you focus on what to do and how to do it
  - ignoring trivial syntax
  - as third semester programming students, you all probably know how important this technique is by now
- now build from this version, see how to add more advanced features

Use an interface
Objective: now use a Java interface, to separate abstract primitive operators from the actual implementation

Use an interface to separate primitive operators from implementation

- remembering from the CSCI 114 prereq course, we use a Java interface to separate a data structure's abstract primitive operators from the actual implementations in Java
  - e.g. here, could then implement the stack either statically using an array, or dynamically using references, and the program using the stack will not know or care
  - excellent!
- will add a Java interface to our simple dynamic implementation of stack

The `StackInterface` interface

- remember, the job of an interface is to declare methods that classes implementing the interface must implement
  - so the `push()`, `pop()` and `isEmpty()` primitives here:

```
public interface StackInterface
{
    void push(int i);

    int pop();

    boolean isEmpty();
}
```

- remember, see, that methods in an interface must be 'abstract'
  - i.e. consist of the method header only, specifying parameters and return types. No implementations allowed here
  - declaring these methods as `public abstract` is redundant and should be avoided

Implementing the `StackInterface` e,g, `LinkedStack`

- now classes must be written that implement all of the methods declared in the interface

- e.g. for the linked list implementation of the StackInterface:

```
public class LinkedStack implements StackInterface
{
    . . .
}
```

- e.g. then for an array implementation of the StackInterface could be:

```
public class ArrayStack implements StackInterface
{
    . . .
}
```

- remember that there is the 'is-a' relationship between an interface and its subclasses, that:
  - `LinkedStack` is-a `StackInterface`
  - `ArrayStack` is-a `StackInterface`
- then no further change required to the linked list implementation, other than this different syntax at the class header!

How to use the `StackInterface`

- so client code is written to use the interface, knowing that these methods are provided, no matter how they are actually implemented
  - in `Tester::main()`, see that we use a `StackInterface` superclass reference `s`, and create a new `LinkedStack` object:

```
StackInterface s = new LinkedStack();
```

  - this is fine, since a `LinkedStack` is-a `StackInterface`
  - then remember that polymorphism makes every method call bind to the appropriate implementation of the method. The linked list version here

Summary

- from Canvas, 'Dynamic implementation of stack' module, Example programs, download, read, run and understand my `Use interface` example program
- next, build from this version, see how to make the data type on the stack generic

Make generic
Objective: will now rewrite the classes for a generic data type, so that we can have a data structure for any class!

Make a data structure for a generic data type

- we now use the Java 'generic' feature to make a data structure for a generic data type
  - e.g. change our stack here from a stack hard coded to be integers only, to a stack of any type of object!
  - much more general, cool!
- uses a feature of Java called 'generics'
  - this allows a data type to be parameterized
  - so the implementation is written for a generic data type
  - a specific data type is given when the program runs, just like a parameter
  - then Java generates code for this specific data type from the generic implementation

How to make the `Item` class generic

- here's the syntax to make a class generic, with a parameter for the data type:

```
public class Item<T>
{
    protected T info;
    protected Item<T> next;
```

  - `T` is used by convention for the name of the data type parameter
  - so see that the name of the class is now `Item<T>`, showing that it is a generic class
  - now the data type of `info` is `T`
  - so `next` now becomes a reference to an `Item<T>` object
- then see the rest of the class for how `T` is used in the methods:

```
    public Item()
    {
        info = null;
```

```
        next = null;
    }

    public Item(T info)
    {
        this.info = info;
        next = null;
    }
}
```

– note that the correct header for a constructor is `public Item()`, not `public Item<T>`

Make `StackInterface` generic

- now the classes using `Item<T>` must also be rewritten to be generic. Simple changes. Here's `StackInterface<T>`:

```
public interface StackInterface<T>
{
    void push(T element);

    T pop();

    boolean isEmpty();
}
```

Then `LinkedStack<T>` implementation

- then the linked list implementation of the interface, `LinkedStack<T>`

```
public class LinkedStack<T> implements StackInterface<T>
{
    private Item<T> top;
```

- for example, see the new syntax of `push()`:

```
    public void push(T element)
    {
        Item<T> item = new Item<T>(element);
```

– remember that `push()` takes a parameter for the new element to be pushed to the stack

– so data type is the generic `T` here

– see how to create the new object named `item` of this generic data type `T`

– then this is placed at the front of the list as usual

- and so on

<u>How to set the generic data type</u>

- we have to set the generic data type when we create an object from a generic class e.g. in `Tester::main()`:

```
StackInterface<Character> s = new LinkedStack<Character>();
```

  - see how the required data type is passed like a parameter!
  - can be any class, so we can have a stack of `Integer`, `Double`, `Character`, `Car`, `Ship`, anything
  - yet we write only one single generic implementation!
  - Java invisibly generates the necessary code from our generic implementation, as required
  - this is really, really cool!

<u>Summary</u>

- from Canvas, 'Dynamic implementation of stack' module, Example programs, download, read, run and understand my `Make generic` example program
  - see in `Tester::main()` that I made a stack of `Character` for a change, then used the values 'A' .. 'F' in the loop. Just for fun
- next, let's add exception handling to this version

Add exceptions
Objective: finally, will add Java exception handling to this version

Use Java exceptions to handle errors

- remember from the CSCI 114 prereq course that, typically, we use Java exception handling to respond to errors that are likely to happen as the program runs
  - for example, what if we pop a stack that is empty?
- remember the 3 parts to exception handling, as we shall see here:
  - create a custom exception class for the error
  - test for the error and throw the exception object when it has happened
  - `try … catch` surrounds the code where the error may occur

The custom `StackUnderflowException` class

- create an appropriate custom exception class for the error
  - we typically use the Java library `RuntimeException` class for this
  - (BTW, is an unchecked exception, we are not actually required to handle it)
  - just add simple, standardized code in our custom class:

```
public class StackUnderflowException extends RuntimeException
{
    public StackUnderflowException()
    {
        super();
    }

    public StackUnderflowException(String message)
    {
        super(message);
    }
}
```

Throw the exception object

- we create and throw the custom exception object in the method where the error occurs. So `LinkedStack::pop()` here:

```
    public T pop() throws StackUnderflowException
```

```
    {
        if (isEmpty())
            throw new StackUnderflowException
                        ("Pop attempted on empty stack");
        else {
            . . .
```

- create and throw the exception object if we try to pop an empty stack

- good idea to declare the throw in the method header as here, for clarity. But not syntactically required

- and would do the same in `StackInterface<T>` here

<u>Try the risky operation, catch and handle if necessary</u>

- now we know that pop is risky, that it may cause a stack underflow error. So we use the Java `try … catch` syntax to try it, catch and handle the error if necessary e.g. in `Tester::main()`:

```
StackInterface<Integer> s = new LinkedStack<Integer>();

// deliberate underflow
try {
    System.out.println(s.pop());
}
catch (StackUnderflowException underflow) {
    System.out.println("Exception: " +
                              underflow.getMessage());
    System.err.println("Exception: " +
                              nderflow.getMessage());
    System.exit(1);
}
System.out.println("done");
```

  - we try the risky pop operation, and most of the time it will work fine, the program will continue running after any catch blocks as normal

  - but sometimes the stack is empty as here, deliberately, so pop will throw an exception. The exception is caught, error messages are output everywhere, and the program halts

- see how exception handling is clearly separated from the ‘normal’ code

  - so the programmer can focus first on the normal situations that happen most of the time. Exactly what we want

  - after the normal code is all written and tested, can then come back and add the separate code that handles the exceptions that don't happen often

  - this exception handling code is separate, does not obscure the structure of the normal code

<u>Summary</u>

- from Canvas, 'Dynamic implementation of stack' module, Example programs, download, read, run and understand my `Add exceptions` example program

  - see in `Tester::main()` that we deliberately pop an empty stack

- we will use this final, more sophisticated list implantation of stack in the exercise assigned this week

Next week and homework

- next week: stack applications

- homework by next week:

  – download, run and understand this week's example programs

  – read Horstmann, section 15.6 on stack applications

Lab

- exercise assigned this week (is not collected or graded) / reviewed next week