C PROGRAMMING III

Ruben Acuña

Spring 2019

MODULE OVERVIEW

- Learning Objectives:
  - To understand how the preprocessor works in C.
  - Apply multiple source files, the preprocessor, and the linker to construct modular programs.
  - Apply procedural programming techniques to perform object-oriented programming.
    - Describe the basic concepts of objects.
    - Explain how an object may be represented in a procedural language.
    - Develop the outline for an object-oriented ADT in a procedural language.
    - Implement a header file for representing an ADT from an outline.
    - Implement functions for constructing and destructing an ADT from an outline.
    - Implement functions for supporting an ADT's behavior from an outline.

  - Apply a file format specification and a file I/O library to read and write binary files.
    - Describe the concept of binary data, and how it is different from text data.
    - Describe the basic standard library functions for manipulating binary data.
    - Explain the structure and format of data indicated by a binary file format.
    - Develop a binary file format based to represent a struct.
    - Implement a function for saving the state of a struct using a binary file format.
    - Implement a function for loading the state of a struct using a binary file format.

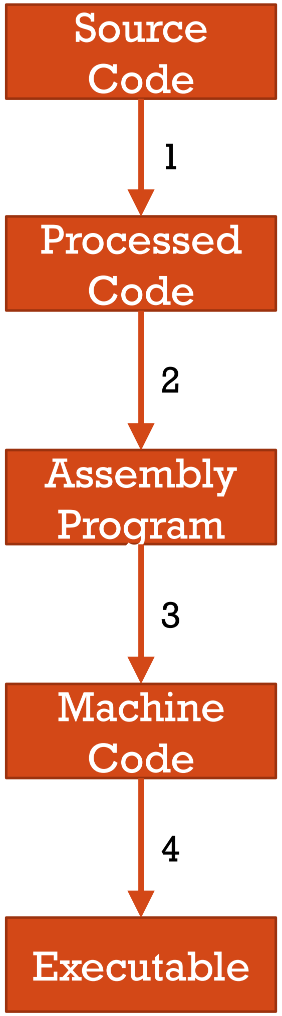
3

THE PREPROCESSOR

C COMPILATION PROCESS

The process to create an execute from a C program is a little involved:

- 1: Preprocessor (new!)
- 2: Compiler
- 3: Assembler
- 4: Linker



PRE-PROCESSING: MACROS

So what happens during preprocessing?

- Macro processing substitutes the definition (code) for the name in the program. Arguments are allowed, creating "in-line" functions / procedures.
- Macro eliminates invocation (function call) overheads.
- Macro processing happens in the pre-processing phase before any compilations or interpretations.

This is called a preprocessor *directive*.

```
#define MAXVAL 100

#define QUADFN(a,b) a*sqrt(b)+b*b - 2*a*a*t++

x = MAXVAL + QUADFN(5,16);
     ⇓
x = 100 + 5*sqrt(16) + 16*16 - 2*5*5*t++;
```

MACROS VS. FUNCTIONS/PROCEDURES

C Function Example:

```
int abs(int a) {
       return ((a < 0) ?  -1 * a : (a));
}
void main() {
       i   = 3;
       j = abs(++i);
       printf(“j = %d”, j);
}
```

4 will be passed to abs function. The result is 4.

MACROS VS. FUNCTIONS/PROCEDURES

C Macro Example:

```
#define abs(a)  ((a < 0) ?  -1 * a : (a))

void main() {
        i   = 3;
        j = abs(++i);
        printf(“j = %d”, j);
}
```

Compiler specific!
BAD BAD

In this example, the result is 5. What happened?

As a side note, ever
hear of inline functions?

A macro call will simply replace like a rubber stamp.   Due to the nature of simple textual replacement, a macro may cause side-effects.

USEFUL DIRECTIVES

- There are quite a few directives we can use.
- Here are a few of the common ones “in the wild”:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include “utilities.h”


#pragma warning(disable: 4996)
…
```

Defines a symbol.

```
#ifndef UTILITIES_H
#define UTILITIES_H

#include <stdio.h>

void flush(FILE *std);
…

#endif UTILITIES_H
```

A symbol automatically defined on windows.

```
#ifdef _WIN32
#include <window.h>

#elif __OSX__
#error Only real computers supported.

#else
#include <unistd.h>

#endif
```

PREDEFINED MACRO NAMES

- In the previous slide, a #ifdef checked if _WIN32 was defined.
- In general, compiles define some set of symbols. Some are just defined, others have values.
- Others include:
  - __DATE__
  - __FILE__
  - __LINE__
  - __TIME__

MODULAR PROGRAMMING

MODULES

- The problem: programs can be complex, with many functions. Does it make sense to put everything in one? Probably not!
- Need to group programs into units larger than functions/procedures; ideally want to group both functions and data.
  - Conceptual reasons (data abstraction)
  - Sharing (library functions)
  - Separate compilation (maintenance issue)
- A **module** usually consists of
  - Specification: says what the module does; gives external view (in Java term: *interface*);
  - Implementation: implementation part (duh).
- Identifiers (functions, variables) given in the *specification* part are available to users of the module and to the implementation part.
- Identifiers given in the *implementation* part are *not* available to external users.

STACK EXAMPLE

- Consider implementing a program that uses a stack to reverse a string and display it. We can try to break it down into two or three major parts:
    - User interaction
    - Stack ADT
    - Console utilities (just for example)
- One approach to making this program more modular is to separate the user interaction code and stack into separate modules.
- This will involve creating a **header file** (.h) and an **implementation** (.c) for each of these proposed modules.
- Each module should contain all the code that is relevant to it. For example, the stack file(s) will contain the two main functions that manipulate a stack as well as the structure that represents stacks. The console utilities file(s) will then contain helper functions related to using the console.

SELECTING MODULES

- Goal: Keep related code together.
- Not necessary to create a separate file for a struct - its functionality is naturally grouped with the stack manipulation functions.

- Modules are intended to structure a program to improve development – increase programmer comprehension and compiler efficiency.
  - Creating unnecessary modules work against this purpose. Too many modules can create overhead for the programmers and the compiler.

- In general, start constructing modules for regions of a program when you have several hundred lines of code.
- When a module itself starts to become lengthy, examine it's contents to determine if it can be refactored into some smaller modules.

HEADER FILES

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

//...

void branching(char);
void* pop(struct stack* s);
void push(struct stack* s, void* element);
void flush();

void main() {    //...
}

void flush() {
    int c;
    do {
        c = getchar();
    } while (c != '\n' && c != EOF);
}
```

- Header files contain only the specifications for some code.
- Here, that means the forward declaration for the flush() function together with any standard include files that the function relies on.

```
// Utilities.h

#ifndef UTILITIES_H
#define UTILITIES_H

#include <stdio.h>

void flush(FILE *std);

#endif UTILITIES_H
```

HEADER GUARDS

```c
// Utilities.h

#ifndef UTILITIES_H
#define UTILITIES_H

#include <stdio.h>

void flush(FILE *std);

#endif UTILITIES_H
```

- The header file we previously produced had some extra preprocessor code.
- This pattern of ifndef / define / code / endif is called a header guard.
- It ensures that if a header file is included multiple times (via the #include directive), the source code will only ever be compiled once.
- This prevents the compiler from stopping due to ‘redeclaration errors’.

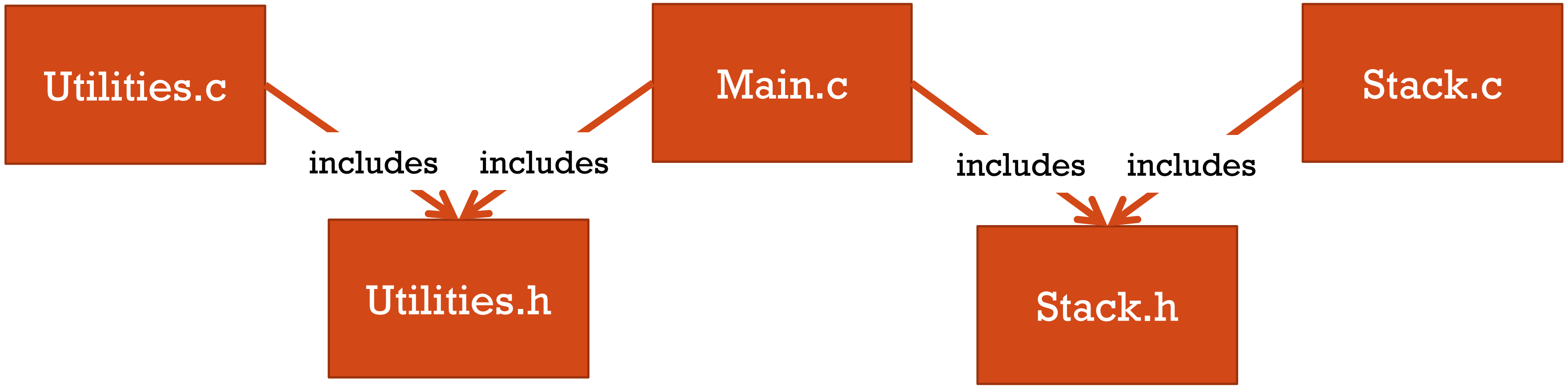
IMPLEMENTATION FILES

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

//...

void branching(char);
void* pop(struct stack* s);
void push(struct stack* s, void* element);
void flush();

void main() {   //...
}

void flush() {
    int c;
    do {
        c = getchar();
    } while (c != '\n' && c != EOF);
}
```

- Implementation files contain only the implementation for some code.
- Here, that means the definition for the flush() function.
- Must also include the header file that was created for the flush function. Quote the filename to do this.

```
//Utilities.c
#include "Utilities.h"

void flush() {
    int c;
    do {
        c = getchar();
    } while (c != '\n' && c != EOF);
}
```

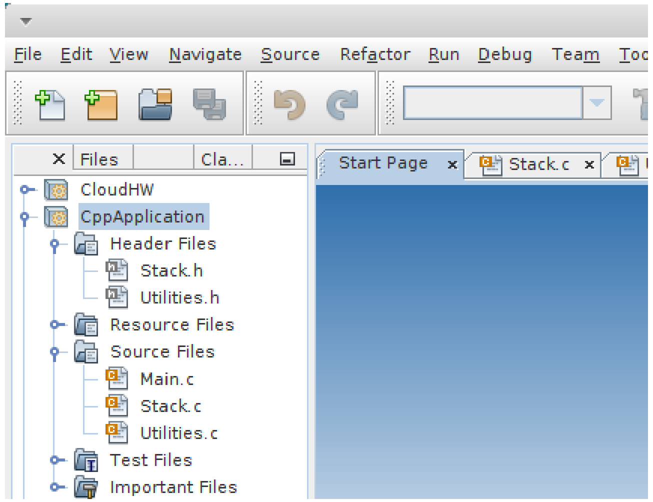
PROJECT LAYOUT



- By following these guidelines the project will be set up so the code for the stack and the console utilities exist in separate files.
- In order to use the functionality from the other modules in the main file (Main.c), it must use the #include directive to access the newly created header files.

COMPILING MODULES

- NetBeans requires that module files be added to the project.
- This involves either creating the files directly in the project or copying the files into the project folder and dragging them into the file.

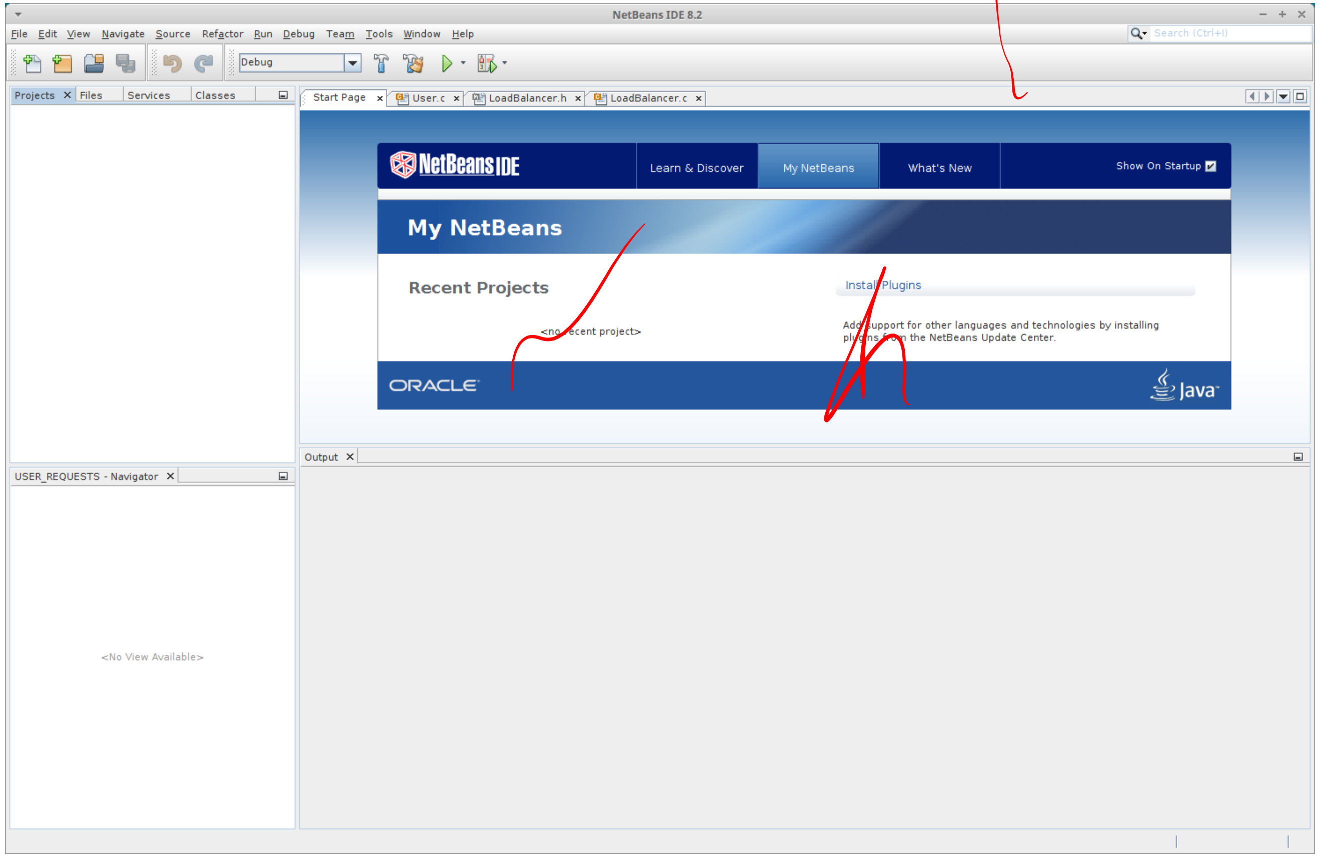


- Using GCC at command-line requires a special command to be used to build the source code:

```
gcc Main.c Utilities.c Stack.c -o output
```

- This indicates that the input to GCC (compiler, and linker) is two .c files which will be compiled into a single output.

IMPLEMENTING MODULES IN NETBEANS



My NetBeans

Recent Projects

<no recent project>

Install Plugins

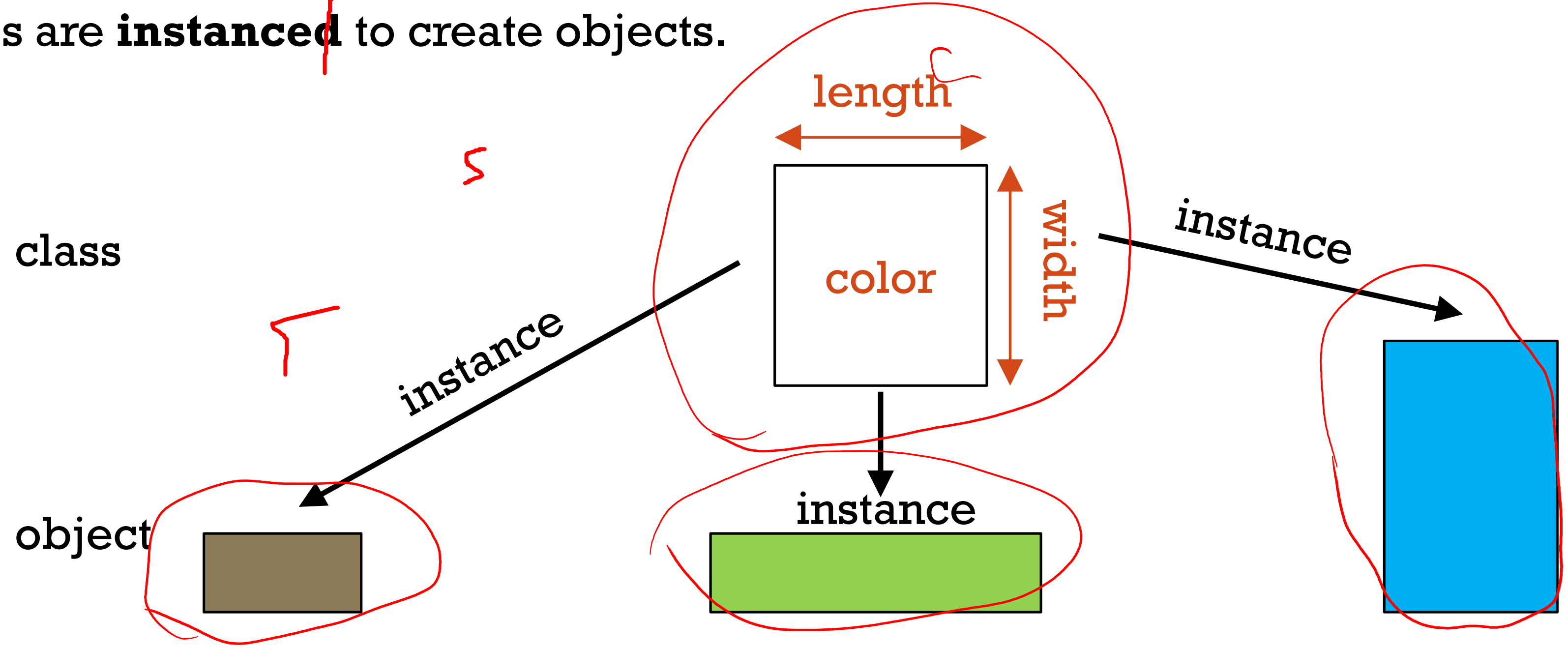
Add support for other languages and technologies by installing plugins from the NetBeans Update Center.

<No View Available>

OBJECT-ORIENTED PROGRAMMING

THE OBJECT CONCEPT

- The moment you didn't know you were waiting for, the return to object land!
- The idea of an **object** is a programming construct that contains both data (*state*) and functions (*behavior*), and functions may operate on the data.
- **Classes** are a blueprint for creating objects that define the data, and behavior, they contain.
- Classes are **instanced** to create objects.



REPRESENTING OBJECTS IN C

- When you previously used objects in Java or C++, it was straightforward since the language supported it out of the box.
- Not so with C!

- We have to understand three key ideas for the implementation of object systems:
  - Storing state.
  - Constructing (or destructing) objects.
  - Invoking behavior on specific objects.
- How did we do these things in Java?

CLASS{
INT a;
INT b;
};

NEW RECT

.

DOT OPERATION

REPRESENTING OBJECTS IN C

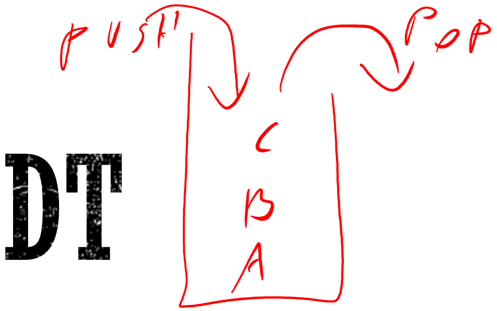
- How can we support the three key ideas of a object system?

- Storing state:
  - Use a struct – it's the natural way to combine pieces of data in C. Only problem, no way to add in functions (which would give us a object system).

- Constructing (or destructing) objects:
  - Use a function that returns a newly created struct that presents the state of the object on creation. Also have a function which takes that struct as an argument and destructs it.

- Invoking behavior on specific objects:
  - A little tricky: use a function which takes a object struct as input plus the normal parameters that the function would need. This means the function will the needed data context to work. (The code inside of the function will be able to access struct data like methods access instance variables.)

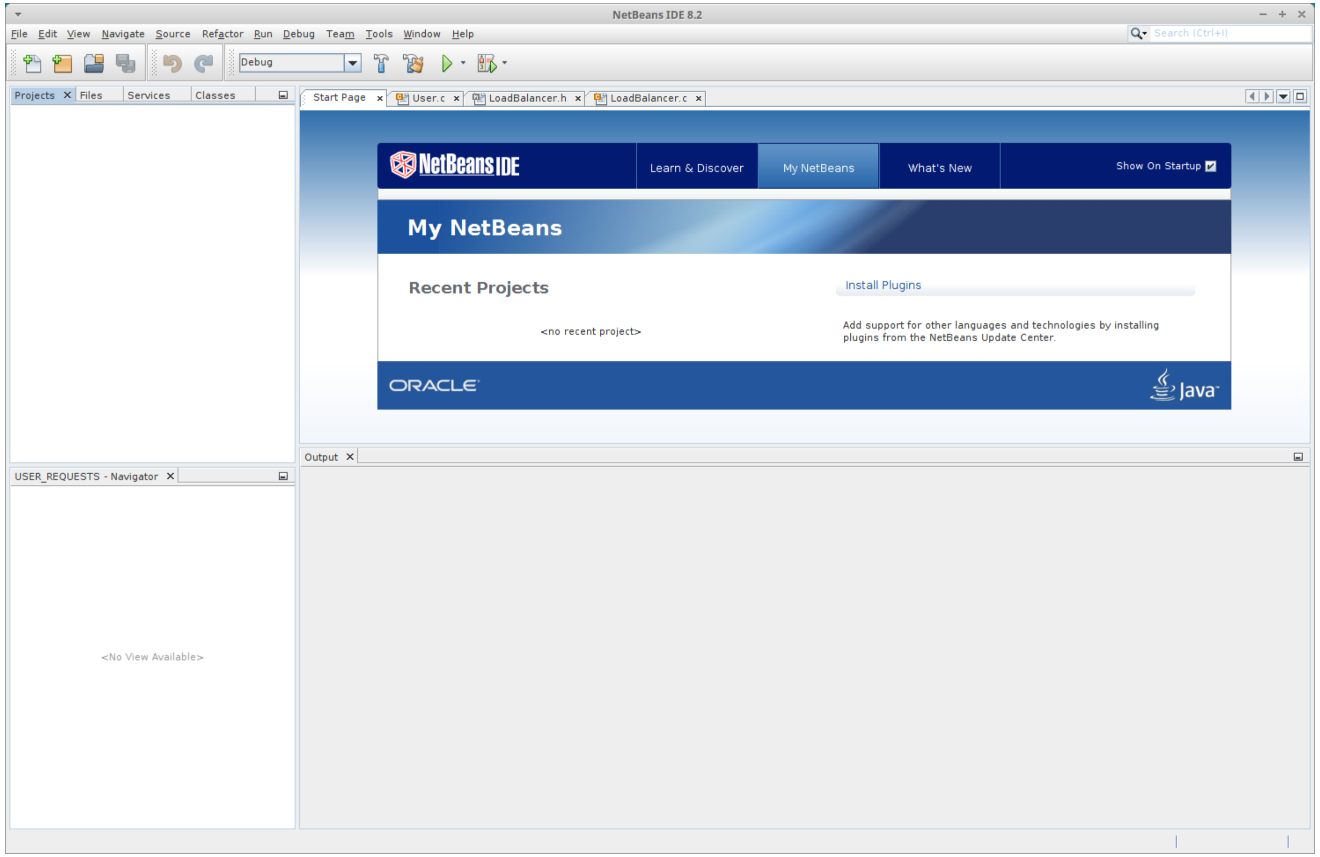
DESIGNING A STACK ADT



- So taking what we know about representing an object in in C, let's try outlining a design for one.
- Let's go for creating a stack ADT. We need to take care of:
  - Storing its state.
  - Constructing (or destructing) it.
  - Invoking behavior on it.
- Where is the "class" here?
- Where is the "instance" here?

```
CLASS {
struct stack{
     INT SIZE;
     NODE* HEAD;
     //SOMETHING ELSE
}

//purpose: creates a new stack and returns it
//return: a pointer to a new stack
STACK* stack_create()

//purpose: destroys a stack
//return: n/a
VOID stack_destroy(STACK* S)

//purpose: adds an element to the stack
//return: n/a
VOID stack_push(STACK* S, INT e)

//purpose: returns most recently added stack element
//return: the element that has been removed
}  INT stack_pop (STACK* S)
```

IMPLEMENTING A STACK SPECIFICAITON

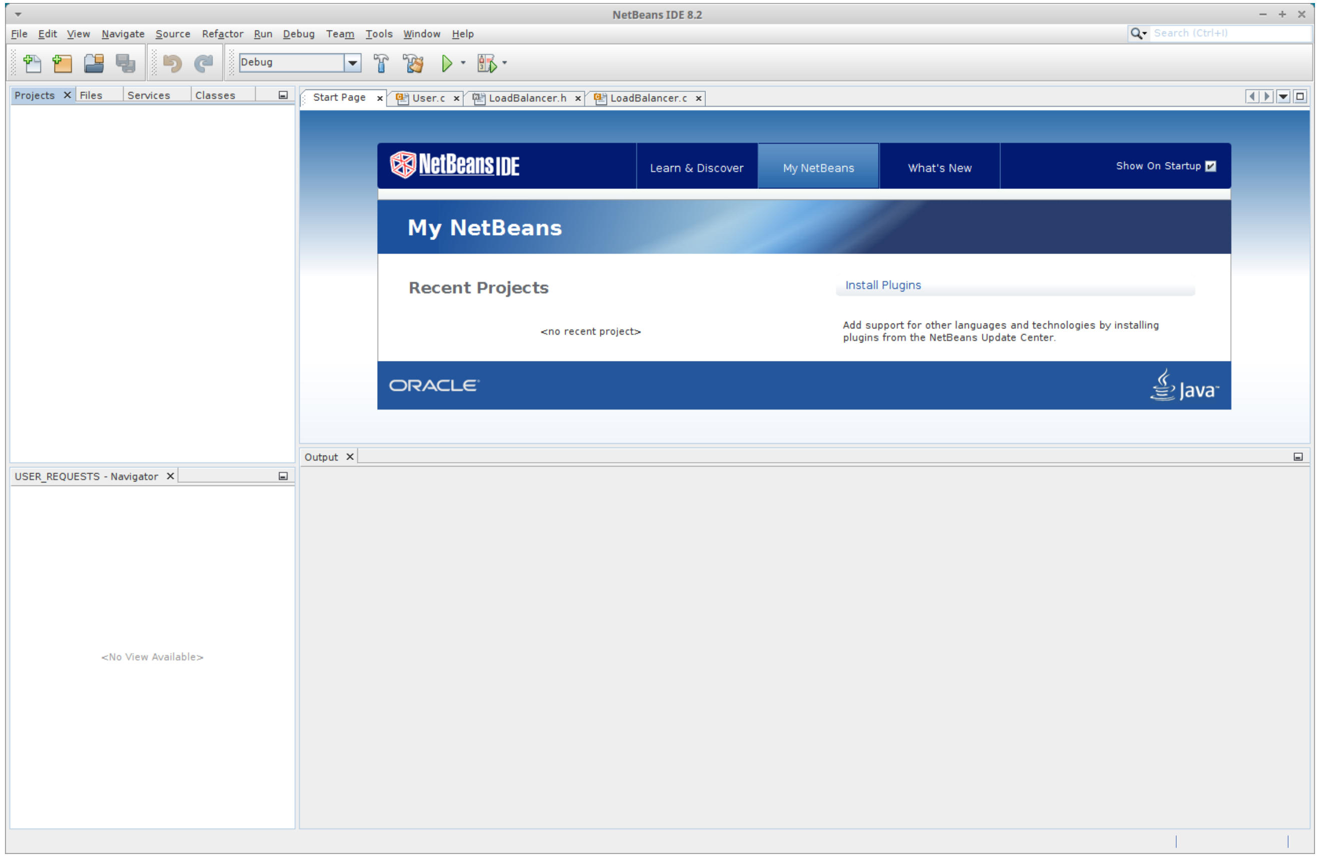


<no recent project>

Add support for other languages and technologies by installing plugins from the NetBeans Update Center.

<No View Available>

IMPLEMENTING STACK CORE

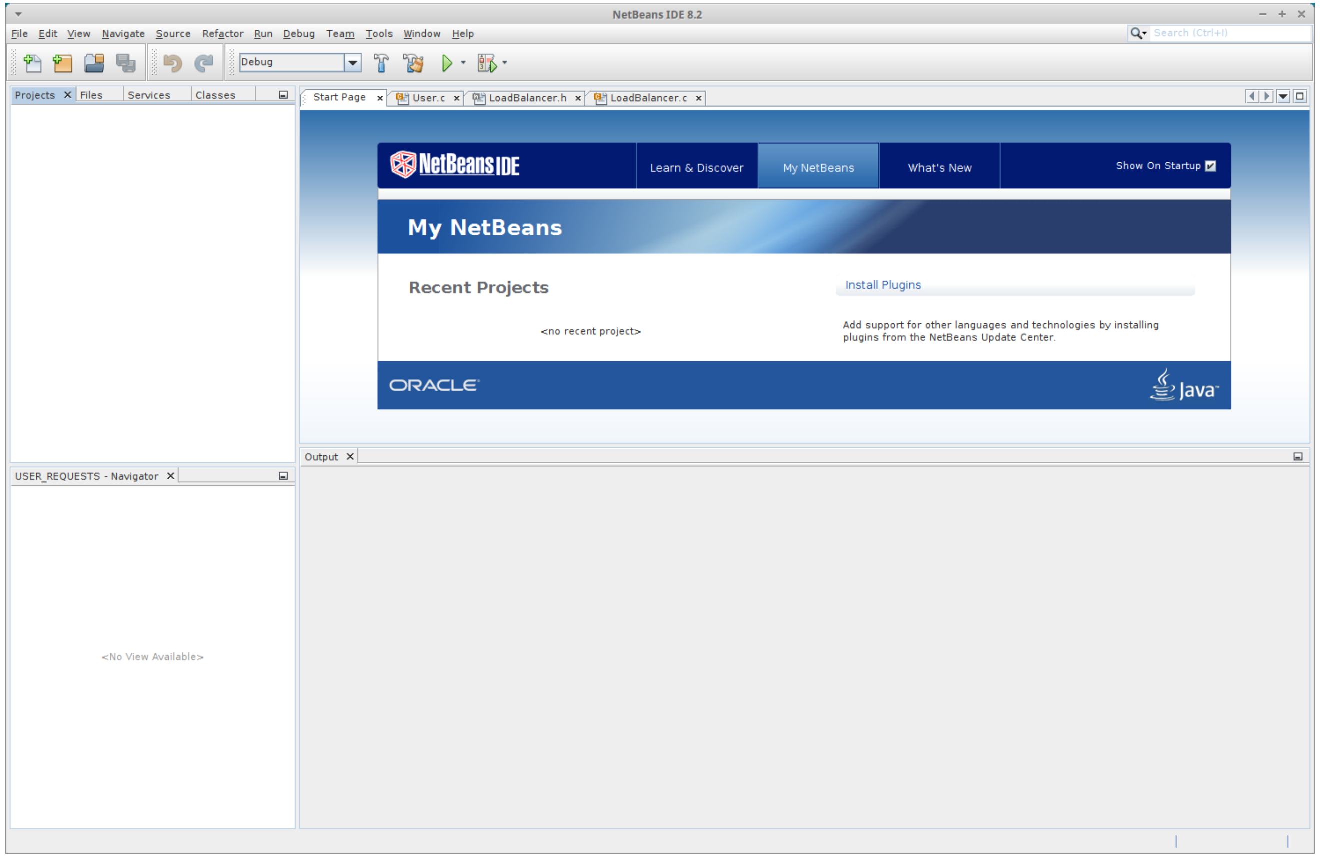

NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Team Tools Window Help
Search (Ctrl+I)
Debug
Projects Files Services Classes
Start Page User.c LoadBalancer.h LoadBalancer.c
NetBeans IDE
Learn & Discover
My NetBeans
What's New
Show On Startup
My NetBeans
Recent Projects
Install Plugins
ORACLE
Java
Output
USER_REQUESTS - Navigator

<no recent project>

Add support for other languages and technologies by installing plugins from the NetBeans Update Center.

<No View Available>

IMPLEMENTING STACK BEHAVIOR


NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Team Tools Window Help
Search (Ctrl+I)
Debug
Projects Files Services Classes
Start Page User.c LoadBalancer.h LoadBalancer.c
NetBeans IDE
Learn & Discover
My NetBeans
What's New
Show On Startup
My NetBeans
Recent Projects
<no recent project>
Install Plugins
ORACLE
Java
USER_REQUESTS - Navigator
<No View Available>
Output

Add support for other languages and technologies by installing plugins from the NetBeans Update Center.