CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

Today

- Object-Oriented Programming Concluded
- Stacks, Queues, and Priority Queues as Abstract Data Types
- Reference types: Basic Principles of References/Pointers
- String type as a reference type
- Array resizing

Next Time

- Queues continued: Implementing a Queue with a Ring (Circular) Buffer

Computer Science

Object-Oriented Design

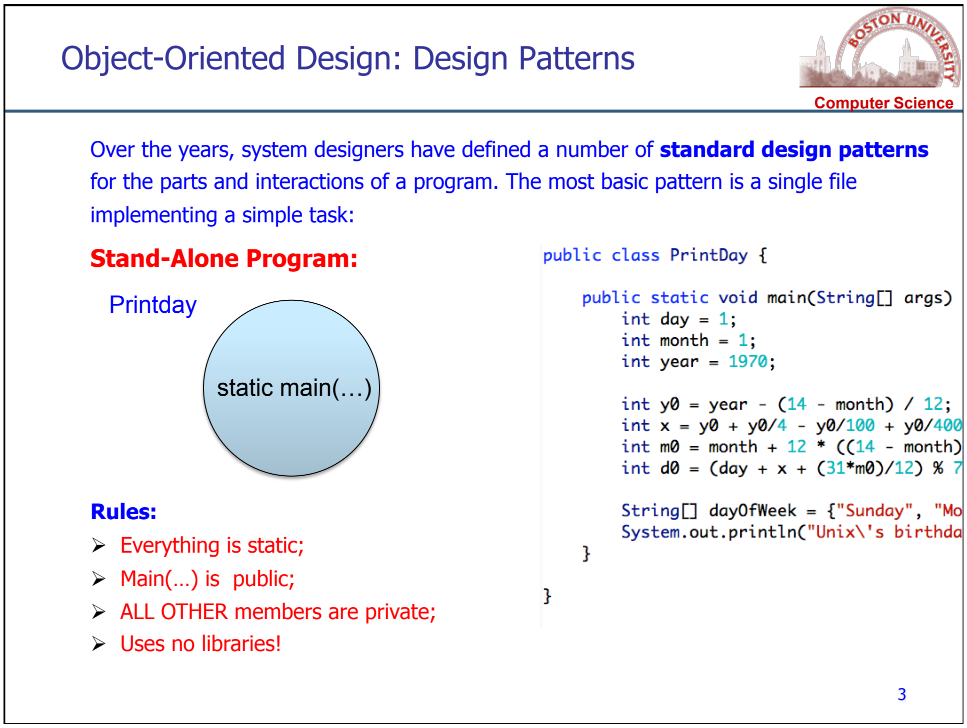
Computer Science

Summary: The most important things to remember about Object-Oriented Design are:

- Divide up your problem and its solution into parts (=classes & objects).
- When you divide, make the interactions (method calls and field references) as simple and easy to understand as possible;
- Make the interface follow KISS -- **provide as few public methods as possible;**
- Use **Information Hiding**: Hide as much about your implementation as you possibly can. If you are not sure whether to make something public or private, **make it private;**

The **advantages of information hiding** are:

- Your code is easier to **understand**, and hence to **use**, and **reuse**;
- Users can't get used to "back-door" ad-hoc features of your code;
- By separating the (simple) **behavior** of your system from the messy details of its **implementation**, you can **change the actual implementation** any time you want---as long as it behaves the same, this is a huge advantage for **maintenance and reuse.**



Object-Oriented Design: Design Patterns

Computer Science

Over the years, system designers have defined a number of **standard design patterns** for the parts and interactions of a program. The most basic pattern is a single file implementing a simple task:

Stand-Alone Program:

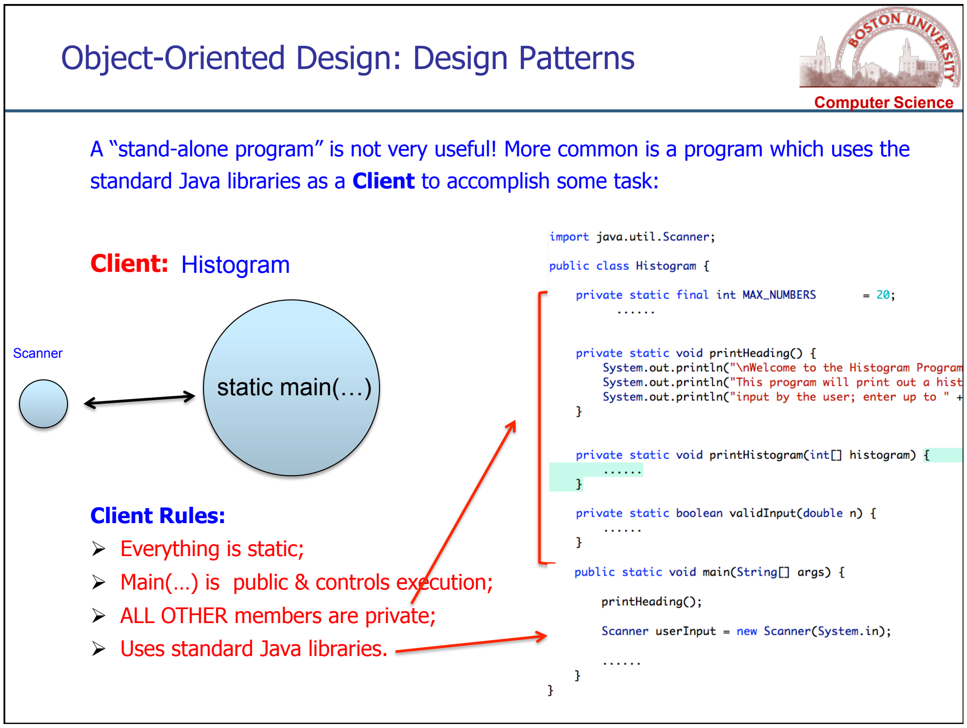
Printday

static main(…)

Rules:

- Everything is static;
- Main(…) is public;
- ALL OTHER members are private;
- Uses no libraries!

```
public class PrintDay {

    public static void main(String[] args)
        int day = 1;
        int month = 1;
        int year = 1970;

        int y0 = year - (14 - month) / 12;
        int x = y0 + y0/4 - y0/100 + y0/400
        int m0 = month + 12 * ((14 - month)
        int d0 = (day + x + (31*m0)/12) % 7

        String[] dayOfWeek = {"Sunday", "Mo
        System.out.println("Unix\'s birthda
    }

}
```



Object-Oriented Design: Design Patterns

Computer Science

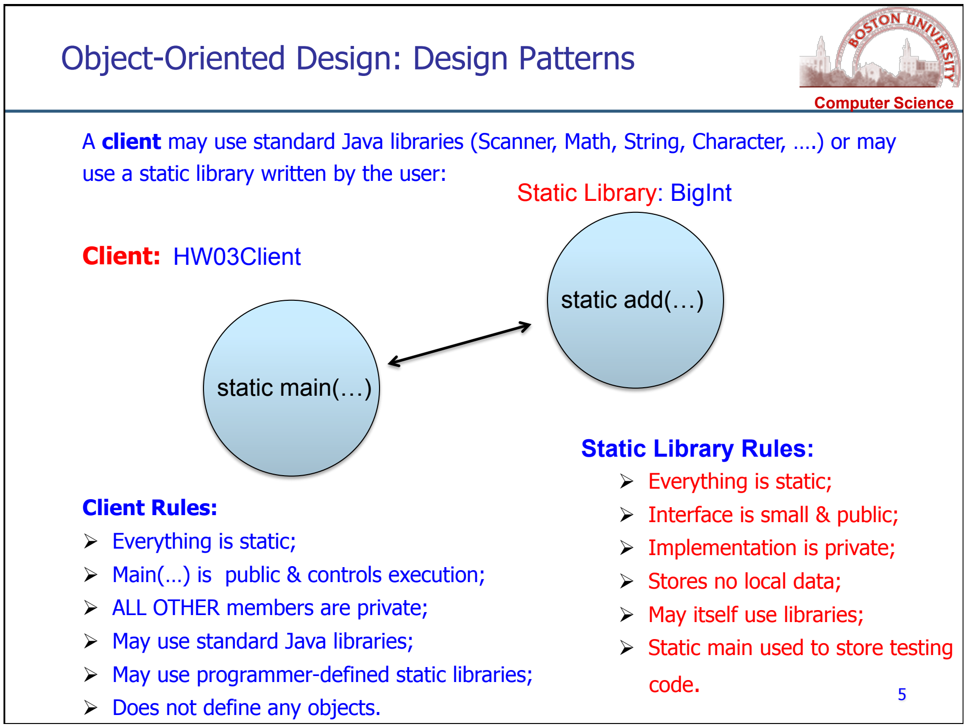
A "stand-alone program" is not very useful! More common is a program which uses the standard Java libraries as a **Client** to accomplish some task:

Client: Histogram

Client Rules:

- Everything is static;
- Main(…) is public & controls execution;
- ALL OTHER members are private;
- Uses standard Java libraries.

```
import java.util.Scanner;

public class Histogram {

    private static final int MAX_NUMBERS       = 20;
         ......


    private static void printHeading() {
        System.out.println("\nWelcome to the Histogram Program
        System.out.println("This program will print out a hist
        System.out.println("input by the user; enter up to " +
    }


    private static void printHistogram(int[] histogram) {
        ......
    }

    private static boolean validInput(double n) {
        ......
    }
   public static void main(String[] args) {

       printHeading();

       Scanner userInput = new Scanner(System.in);

       ......
   }
}
```



Object-Oriented Design: Design Patterns

A **client** may use standard Java libraries (Scanner, Math, String, Character, ….) or may use a static library written by the user:

Static Library: BigInt

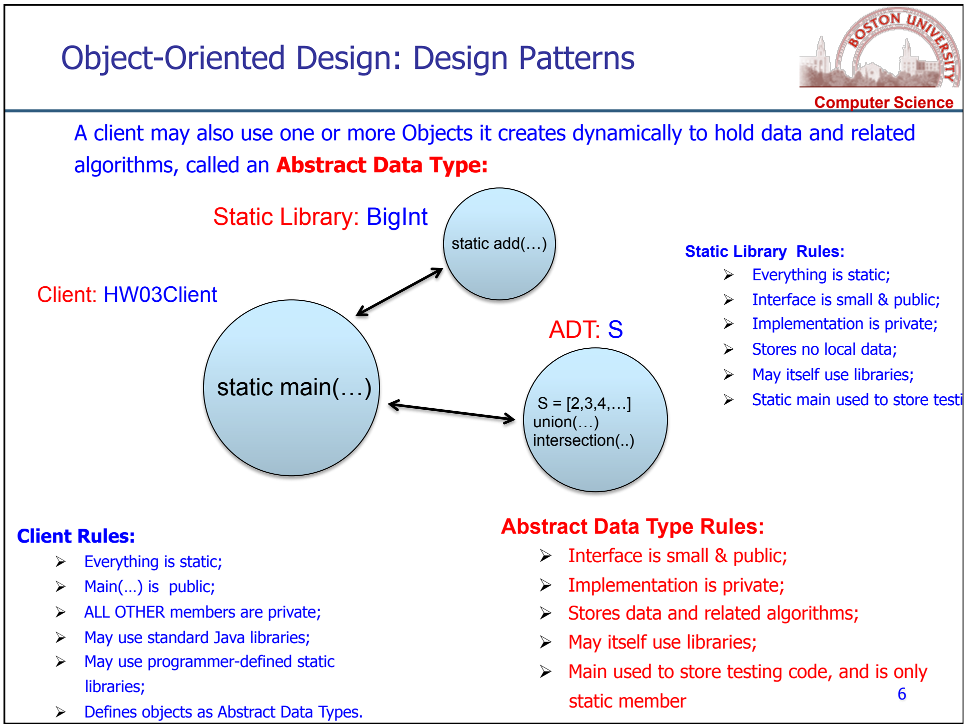
Client: HW03Client

Static Library Rules:

- Everything is static;
- Interface is small & public;
- Implementation is private;
- Stores no local data;
- May itself use libraries;
- Static main used to store testing code.

Client Rules:

- Everything is static;
- Main(…) is public & controls execution;
- ALL OTHER members are private;
- May use standard Java libraries;
- May use programmer-defined static libraries;
- Does not define any objects.



Object-Oriented Design: Design Patterns

A client may also use one or more Objects it creates dynamically to hold data and related algorithms, called an **Abstract Data Type:**

Static Library: BigInt

Client: HW03Client

ADT: S

Static Library Rules:

- Everything is static;
- Interface is small & public;
- Implementation is private;
- Stores no local data;
- May itself use libraries;
- Static main used to store testi

Client Rules:

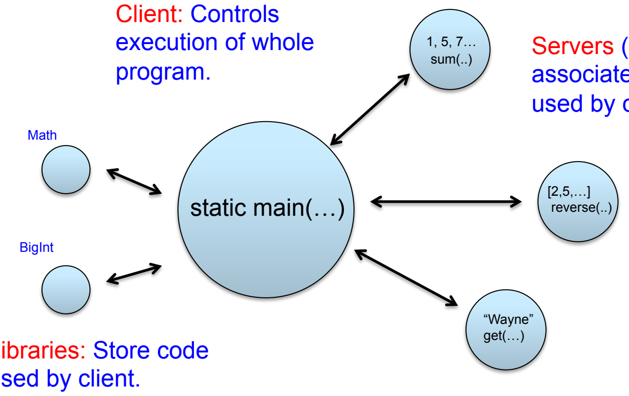
- Everything is static;
- Main(…) is public;
- ALL OTHER members are private;
- May use standard Java libraries;
- May use programmer-defined static libraries;
- Defines objects as Abstract Data Types.

Abstract Data Type Rules:

- Interface is small & public;
- Implementation is private;
- Stores data and related algorithms;
- May itself use libraries;
- Main used to store testing code, and is only static member

Object-Oriented Design: Design Patterns

Sometimes this is called the **Client/Server Model:**



Client: Controls execution of whole program.

Servers (store data with associated algorithms used by client)

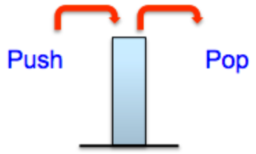
Libraries: Store code used by client.

Abstract Data Types: The Stack ADT

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

Stack Interface (informal):

```
void Push(int n) – Put integer n on top of the stack
int Pop() – Remove top integer and return it
int Peek() – Return the top integer without removing it
int size() – Return the number of integers in the stack
boolean isEmpty() – Returns true iff stack has no members
```



Abstract Data Types: The Stack ADT

Computer Science

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

Stack Interface:

```
void Push(int n)
int Pop()
int Peek()
int size()
boolean isEmpty()
```

push(5);

5

Abstract Data Types: The Stack ADT

Computer Science

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

Stack Interface:

```
void Push(int n)
int Pop()
int Peek()
int size()
boolean isEmpty()
```

push(5);
push(7);

7
5

Abstract Data Types: The Stack ADT

Computer Science

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

Stack Interface:

```
void Push(int n)
int Pop()
int Peek()
int size()
boolean isEmpty()
```

```
push( 5 );
push( 7 );
push( 2 );
```

2

7

5

Abstract Data Types: The Stack ADT

Computer Science

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

Stack Interface:

```
void Push(int n)
int Pop()
int Peek()
int size()
boolean isEmpty()
```

```
push( 5 );
push( 7 );
push( 2 );
int n = pop();
```

n: 2

7

5

Abstract Data Types: The Stack ADT

Computer Science

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

Stack Interface:

```
void Push(int n)
int Pop()
int Peek()
int size()
boolean isEmpty()
```

```
push( 5 );
push( 7 );
push( 2 );
int n = pop();
int m = pop();
```

n: 2

m: 7

5

Abstract Data Types: The Stack ADT

Computer Science

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

Stack Interface:

```
void Push(int n)
int Pop()
int Peek()
int size()
boolean isEmpty()
```

```
push( 5 );
push( 7 );
push( 2 );
int n = pop();
int m = pop();
int i = pop();
```

n: 2

m: 7

i: 5

5

Abstract Data Types: The Stack ADT

Applications of Stacks:

Reversing an array or a String

Keeping track of nested or recursive structure

Parenthesis Matching

Evaluating an arithmetic expression

Run-time Stack to keep track of method/function calls

[Examples on Board]

Abstract Data Types: The Stack ADT

Applications of Stacks:

Reversing an array or a String
Keeping track of calls waiting
Parenthesis Matching
Expression Evaluation
Run-time Stack to keep track of method/function calls

Problems with Stacks:

Underflow: Trying to pop() or peek() and empty stack! Solution: check if empty before doing a peek or pop!

Overflow: Pushing too many numbers and causing an ArrayIndexOutOfBoundsException!
Solution: Array Resizing…..

Reference types: Data in Computer Memory

Computer Science

To understand the notion of references (also called pointers), we need to understand how computer memory works to organize data:

RAM:

0	2
1	5
2	**13**
3	23
4	-34
5	232
6	2
7	6
8	**3**
9	**10**
10	-78
11	3
12	4
13	5
14	5
15	-1
16	2

Computer instructions say things like:

“Put a 3 in location 8:”

```
RAM[8] = 3;
```

“Add the numbers in locations 8 and 9 and put the sum in location 2:”

```
RAM[2] = RAM[8] + RAM[9]
```

This is why arrays are so common and so efficient: RAM is just a big array!

Access time = about 10^{-7} secs

Reference types: Data in Computer Memory

Computer Science

When you create variables in Java (or any programming language), these are “nicknames” or shortcut ways of referring to locations in RAM:

RAM:

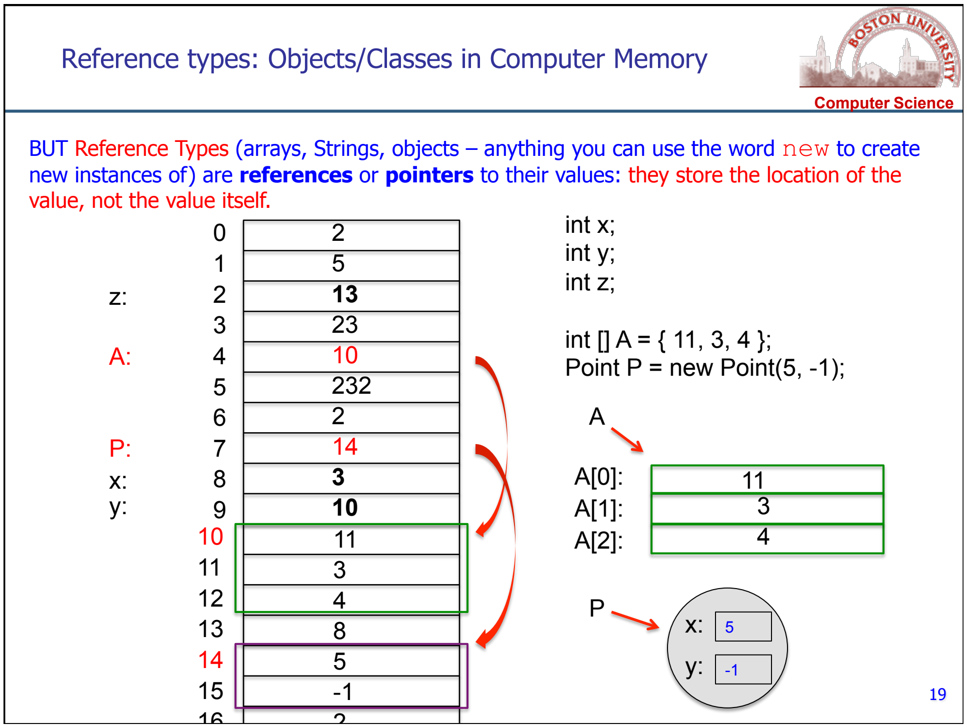
	0	2
	1	5
z:	2	13
	3	23
	4	-34
	5	232
	6	2
	7	6
x:	8	3
y:	9	10
	10	-78
	11	3
	12	4
	13	5
	14	5
	15	-1
	16	2

These “shortcut” names for primitive types can not change during execution.

```
int x;     // same as RAM[8]
int y;     // same as RAM[9]
int z;     // same as RAM[2]

// now the previous computation
// would be

x = 3;
y = 10;
z = x + y;
```

When we draw our diagrams of variables, we are really just giving a shortcut view of RAM without the addresses:

x:	3



Reference types: Objects/Classes in Computer Memory

Computer Science

BUT Reference Types (arrays, Strings, objects – anything you can use the word `new` to create new instances of) are **references** or **pointers** to their values: they store the location of the value, not the value itself.

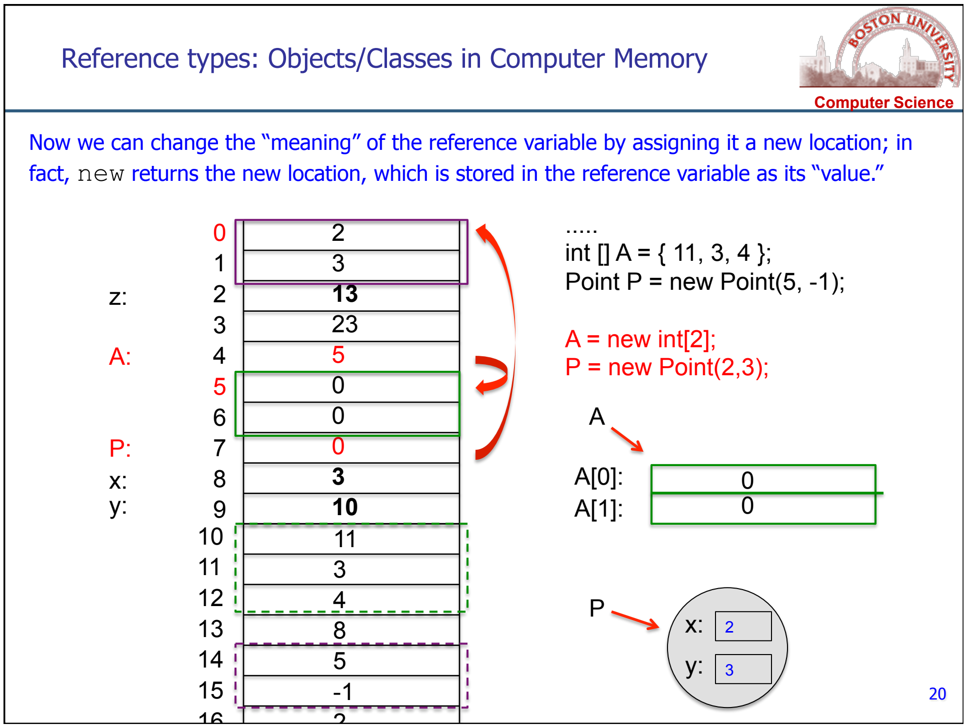
	0	2
	1	5
z:	2	**13**
	3	23
A:	4	10
	5	232
	6	2
P:	7	14
x:	8	**3**
y:	9	**10**
	10	11
	11	3
	12	4
	13	8
	14	5
	15	-1
	16	2

```
int x;
int y;
int z;

int [] A = { 11, 3, 4 };
Point P = new Point(5, -1);
```

A

A[0]: 11
A[1]: 3
A[2]: 4

P



Reference types: Objects/Classes in Computer Memory

Computer Science

Now we can change the "meaning" of the reference variable by assigning it a new location; in fact, `new` returns the new location, which is stored in the reference variable as its "value."

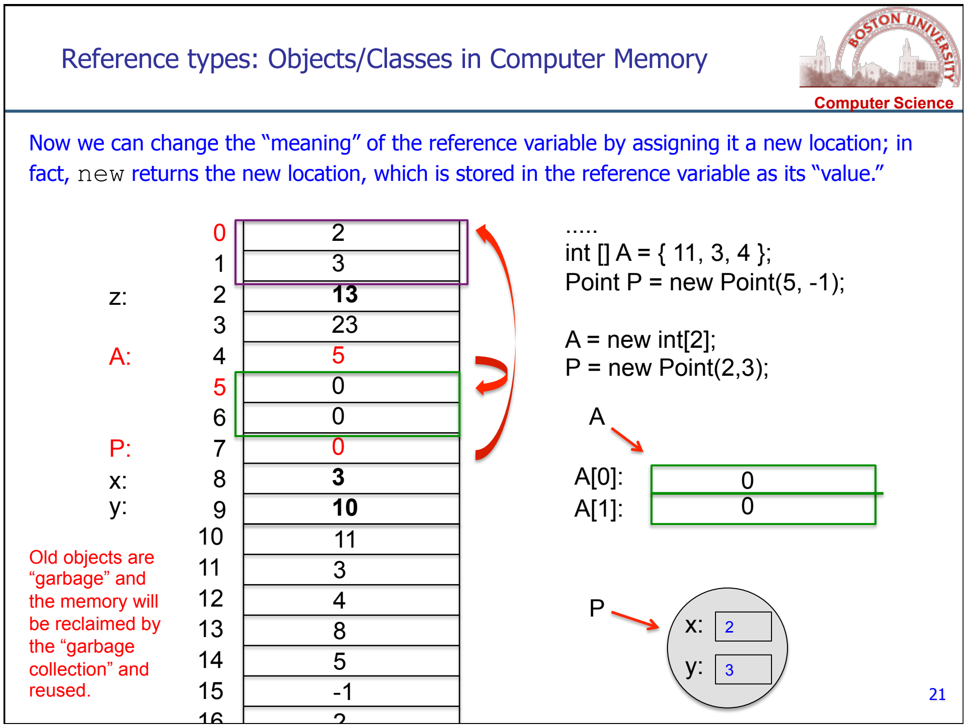
	0	2
	1	3
z:	2	**13**
	3	23
A:	4	5
	5	0
	6	0
P:	7	0
x:	8	**3**
y:	9	**10**
	10	11
	11	3
	12	4
	13	8
	14	5
	15	-1
	16	2

```
.....
int [] A = { 11, 3, 4 };
Point P = new Point(5, -1);

A = new int[2];
P = new Point(2,3);
```

A

A[0]: 0
A[1]: 0

P



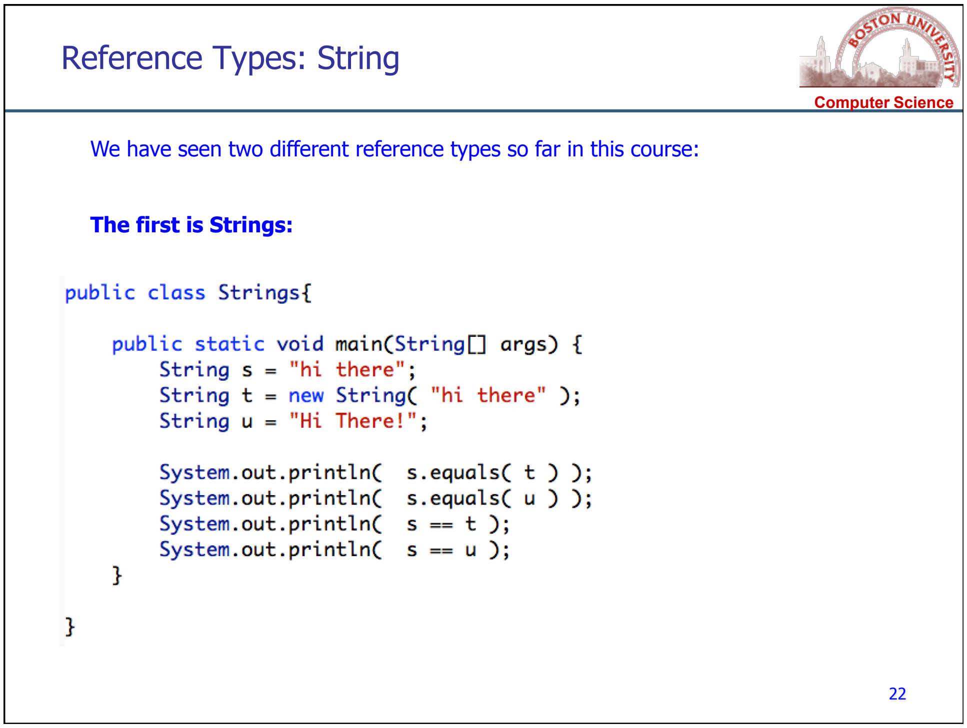
Reference types: Objects/Classes in Computer Memory

Now we can change the "meaning" of the reference variable by assigning it a new location; in fact, `new` returns the new location, which is stored in the reference variable as its "value."

	0	2
	1	3
z:	2	**13**
	3	23
A:	4	5
	5	0
	6	0
P:	7	0
x:	8	**3**
y:	9	**10**
	10	11
	11	3
	12	4
	13	8
	14	5
	15	-1
	16	2

Old objects are "garbage" and the memory will be reclaimed by the "garbage collection" and reused.

```
.....
int [] A = { 11, 3, 4 };
Point P = new Point(5, -1);

A = new int[2];
P = new Point(2,3);
```



Reference Types: String

We have seen two different reference types so far in this course:

The first is Strings:

```
public class Strings{

    public static void main(String[] args) {
        String s = "hi there";
        String t = new String( "hi there" );
        String u = "Hi There!";

        System.out.println(  s.equals( t ) );
        System.out.println(  s.equals( u ) );
        System.out.println(  s == t );
        System.out.println(  s == u );
    }

}
```

Reference Types: String

We have seen two different reference types so far in this course:

The first is Strings:

```
public class Strings{

    public static void main(String[] args) {
        String s = "hi there";
        String t = new String( "hi there" );
        String u = "Hi There!";

        System.out.println(  s.equals( t ) );
        System.out.println(  s.equals( u ) );
        System.out.println(  s == t );
        System.out.println(  s == u );
    }

}
```

```
> run Strings
true
false
false
false
```

equals checks for same structure;

== checks for same reference (pointing to same location).

Reference Types: String

We have seen two different reference types so far in this course:

The second is arrays:

Let’s look at how to solve the problem of stack overflow, using array resizing:

```
// replace S by array twice as big, but with same elements

private void resize() {
    int[] T = new int[ S.length * 2 ];
    for (int i = 0; i < S.length; ++i) {
        S[i] = T[i];
    }
    S = T;
}
```