CSE 1020: Unit 4

Topics: Object abstraction & usage

To do: Read Chapter 4; Lab 4

1

Outline

- **Objects**
- **API and usage**
- **Patterns**
- **(More) usage**
- **Abstraction: Objects, classes & methods**
- **Software engineering example**

2

Outline

- **Objects**
- **API and usage**
- **Patterns**
- **(More) usage**
- **Abstraction: Objects, classes & methods**
- **Software engineering example**

3

Objects

The object abstraction

- In addition to modules (static classes), there is an allied abstraction for grouping related data and operations.
- An object is an abstraction that combines data (attributes) with ways to operate on the data (methods).
  - Now we have additional leeway in customization.

4

Objects

The object abstraction

- In addition to modules (static classes), there is an allied abstraction for grouping related data and operations.
- An object is an abstraction that combines data (attributes) with ways to operate on the data (methods).
  - Now we have additional leeway in customization.

Example

- An object for a stock on the Toronto Stock Exchange (TSX) might have …
- Attributes
  - name
  - price
- Methods
  - update (refresh) the price

5

Objects

Templates and classes

- We can have many similar objects that differ in the data they contain.
  - For example, stocks for BMO vs. TD vs. …
- All these similar objects belong to the same class.
  - For example, the Stock class
- The class serves as a template for the objects.

6

Objects

Templates and classes

- We can have many similar objects that differ in the data they contain.
  - For example, stocks for BMO vs. TD vs. …
- All these similar objects belong to the same class.
  - For example, the Stock class
- The class serves as a template for the objects.
- To deal with a particular individual of the class, called an object instance of the class…
  - Create a new object
  - That is a customized to your specifications
  - For example, a particular stock, say BMO

7

Objects

Templates and classes

- We can have many similar objects that differ in the data they contain.
  - For example, stocks for BMO vs. TD vs. …
- All these similar objects belong to the same class.
  - For example, the Stock class
- The class serves as a template for the objects.
- To deal with a particular individual of the class, called an object instance of the class…
  - Create a new object
  - That is a customized to your specifications
  - For example, a particular stock, say BMO
- Alternatively, we can think of the class as a factory for objects.

8

Objects

Static vs. non-static classes

- Last unit we saw
  - static classes (modules)
  - here the class serves to group conceptually related data and operations
  - There is no capability of creating customized copies (instances) of the class (no objects)
  - We do not create our own customized versions of ToolBox or Math.

Objects

Static vs. non-static classes

- Last unit we saw
  - static classes (modules)
  - here the class serves to group conceptually related data and operations
  - There is no capability of creating customized copies (instances) of the class (no objects)
  - We do not create our own customized versions of ToolBox or Math.
- This unit we encounter
  - non-static classes (object factories)
  - Once again, the class serves to organize related data and operations
  - But now, the we can create customized instances of the class (objects)
  - We do create individual instances of Stock that differ in their attributes.

10

Outline

- Objects
- **API and usage**
- Patterns
- (More) usage
- Abstraction: Objects, classes & methods
- Software engineering example

11

APIs and class use

API anatomy: A second pass

- Last unit we considered APIs for static classes.
- This unit we elaborate by considering non-static classes.
  - Here we can create and manipulate customized instances of a class, i.e., objects.
  - We will exploit the Stock class as an example.

12

APIs and class use

API anatomy: A second pass

- Last unit we considered APIs for static classes.
- This unit we elaborate by considering non-static classes.
  - Here we can create and manipulate customized instances of a class, i.e., objects.
  - We will exploit the Stock class as an example.

Remark

- Static classes do *not* support the creation of customized objects.

13

APIs and class use

API anatomy: Basic terminology

- The term field refers to a piece of information that is maintained about …
  - the class as a a whole.
    - e.g., IO.fillChar.
    - We will see more this unit for non-static classes.
  - particular object instances of a class.

14

APIs and class use

API anatomy: Basic terminology

- The term field refers to a piece of information that is maintained about …
  - the class as a a whole.
    - e.g., IO.fillChar.
    - We will see more this unit for non-static classes.
  - particular object instances of a class.
    - We will see these this unit for non-static classes, e.g, the name of a particular stock created through the non-static class Stock.
    - We did *not* see these last unit, as static classes do not support the creation of objects.

15

APIs and class use

API anatomy: Basic terminology

- The term method will refer an operation that can be performed…
  - in general with respect to the class; such methods *do not* change or examine object instances of the class
    - We saw these last unit for static classes (e.g., Math.sqrt() in the Math class)
    - We will see more this unit for non-static classes.

APIs and class use

API anatomy: Basic terminology

- The term method will refer an operation that can be performed…
  - in general with respect to the class; such methods *do not* change or examine object instances of the class
    - We saw these last unit for static classes (e.g., Math.sqrt() in the Math class)
    - We will see more this unit for non-static classes.
  - On particular object instances of the class
    - We will see these this unit (e.g., get the price of a particular stock)
    - We did *not* see these last unit

17

APIs and class use

API anatomy: Basic terminology

- The term constructor will refer to an operation that can create a particular object instance of a class (e.g., used to create a particular stock).
- Note: We did *not* encounter constructors last unit, as static classes do not support construction of objects.

18

APIs and class use

API anatomy: Structure

- The documentation is divided into 3 main parts

1. **Descriptive paragraph:** Provides a high level overview of the class.
2. **Summary:** A terse specification of the interfaces to the fields, methods and constructors that are available.
3. **Detail:** Provides additional information about the fields, methods and constructors that are available.

19

APIs and class use

API anatomy: The descriptive paragraph

public class **Stock**
extends java.lang.Object

This class encapsulates a stock. It holds attributes relevant to a stock, …

Version:
7.0 (Summer 2010)
Author:
H. Roumani, roumani@cse.yorku.ca

20

APIs and class use

API anatomy: Fields

Field Summary

	char	delimiter This field determines what character is inserted between the stock symbol and its name
	String	name The name of this stock as listed on the exchange.
static	boolean	titleCaseName This field controls the format of the stock name.

APIs and class use

API anatomy: Fields

Field Summary

	char	delimiter This field determines what character is inserted between the stock symbol and its name
	String	name The name of this stock as listed on the exchange.
static	boolean	titleCaseName This field controls the format of the stock name.

- Same as last unit
  - type, symbolic name, terse descriptor.
  - Keyword static specifies that the field pertains to the class as a whole
- New for this week
  - Absence of the keyword static specifies that the field pertains to particular object instances of the class.

22

APIs and class use

API anatomy: Fields

Field Detail

name

public java.lang.String name

The name of this stock as listed on the Exchange. If the name could not be determined (either because no such stock is listed or a Exchange connection could not be established), the name is set to null. (Bad design --this field should not be made public.)

Here we see specification of
- visibility (public)
- if a field is constant (final)
- more descriptive detail

23

APIs and class use

API anatomy: Methods

Method Summary

double getPrice()
Determine the price of this Stock.

void refresh()
Establish a connection with the exchange and update this stock's attributes accordingly.

boolean setDelimiter(char myDelimiter)
Change delimiter of this stock to the passed one.

- Same as last unit
  - signature, return type, terse descriptor.
  - Keyword static specifies that the method pertains to the class as a whole.

24

APIs and class use

API anatomy: Methods

Method Summary

double getPrice()
Determine the price of this Stock.

void refresh()
Establish a connection with the exchange and update this stock's attributes accordingly.

boolean setDelimiter(char myDelimiter)
Change delimiter of this stock to the passed one.

- Same as last unit
  - signature, return type, terse descriptor.
  - Keyword static specifies that the method pertains to the class as a whole.
- New for this week
  - Absence of static specifies that the method is applied to particular object instances of the class.

25

APIs and class use

API anatomy: Methods

Method Detail

getPrice

public double getPrice()

Determine the price of this Stock.
Returns:
the price of this Stock as last obtained from the exchange.

setDelimiter

public boolean setDelimiter(char myDelimeter)

Mutator to change the delimiter of this stock to the passed one...
Parameters:
myDelimiter – the new delimiter character
Returns:
true if the change was made…

Here we see specification of
- visibility (public)
- greater explanatory detail

26

APIs and class use

API anatomy: Constructors

Constructor Summary

Stock()
Construct a default Stock.

Stock(Stock stock)
Construct a copy of the passed Stock.

Stock(java.lang.String symbol)
Construct a Stock having the (capitalized) passed symbol.

- There are no columns for type, return or static.
- The name must be the same as that of the class.
- Parameters may be present to provide information for instantiating an object.
- It is okay for there to be more than one constructor as long as they have different signatures (overloading).

27

APIs and class use

API anatomy: Constructors

Constructor Detail

Here we see specification of
- visibility (public)
- greater explanatory detail

Stock

public Stock(java.lang.String symbol)

Construct a Stock having the (capitalized) passed symbol. The stock attributes are set as per the refresh() method.
Parameters:
symbol - the (ticker) symbol of the stock to construct.

Stock

public Stock(Stock stock)

Construct a copy of the passed stock.
Parameters: stock – the Stock to copy.
Throws: java.lang.RuntimeException –if Stock is null.

28

APIs and class use

Object creation

- Different objects belong to different classes.
- We say that an object is an instance of its class.
- Declaration is the same as that of primitive types

  Stock s;
- To instantiate the object we must have it constructed

  s = new Stock("BMO");

Operator new
- Allocates memory for the object.

Constructor
- Initializes the object.

29

APIs and class use

BTW, don't forget to import type.lib.* to make make these examples work in an actual program.

Object creation

- Different objects belong to different classes.
- We say that an object is an instance of its class.
- Declaration is the same as that of primitive types

  Stock s;
- To instantiate the object we must have it constructed

  s = new Stock("BMO");

Operator new
- Allocates memory for the object.

Constructor
- Initializes the object.

30

APIs and class use

Object creation

- Different objects belong to different classes.
- We say that an object is an instance of its class.
- Declaration is the same as that of primitive types

```
Stock s;
```

- To instantiate the object we must have it constructed

```
s = new Stock("BMO");
```

- Also can combine declaration and construction

```
Stock s = new Stock("BMO");
```

31

APIs and class use

Object creation

- Different objects belong to different classes.
- We say that an object is an instance of its class.
- Declaration is the same as that of primitive types

```
Stock s;
```

- To instantiate the object we must have it constructed

```
s = new Stock("BMO");
```

- Also can combine declaration and construction

```
Stock s = new Stock("BMO");
```

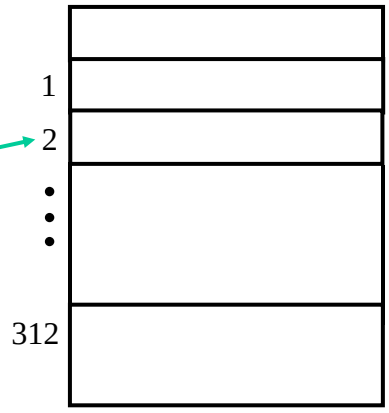
- Distinguish object vs. object reference
  - The object is something in memory
  - The object reference, s, points to the object in memory

32

APIs and class use

Memory diagram: Object declaration and construction

- Upon seeing the declaration
  MyClass mine;
- The processor associates the symbol mine with a memory location.

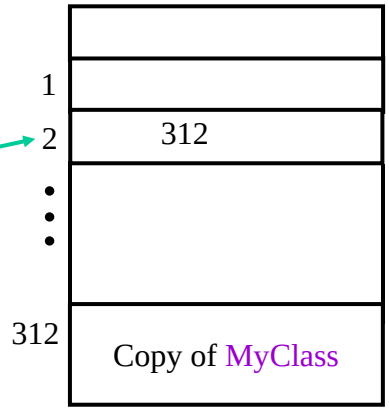


33

APIs and class use

Memory diagram: Object declaration and construction

- Upon seeing the declaration
  MyClass mine;
- The processor associates the symbol mine with a memory location.
- Upon seeing the construction
  mine = new MyClass( );
- The processor makes a copy of the entire class MyClass in memory and loads that memory location into the contents of mine.

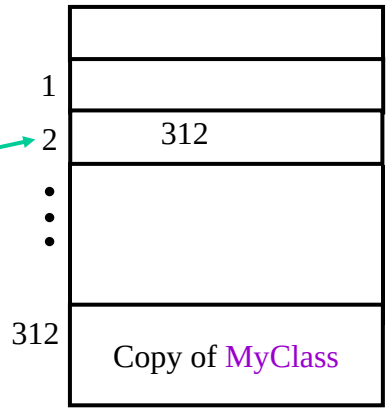


34

APIs and class use

Memory diagram: Object declaration and construction

- Upon seeing the declaration
  MyClass mine;
- The processor associates the symbol mine with a memory location.
- Upon seeing the construction
  mine = new MyClass( );
- The processor makes a copy of the entire class MyClass in memory and loads that memory location into the contents of mine.



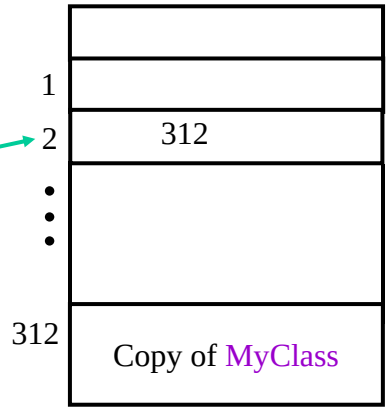
Remark 1: We see that an object is a copy (instance) of the class that resides in memory.

35

APIs and class use

Memory diagram: Object declaration and construction

- Upon seeing the declaration
  MyClass mine;
- The processor associates the symbol mine with a memory location.
- Upon seeing the construction
  mine = new MyClass( );
- The processor makes a copy of the entire class MyClass in memory and loads that memory location into the contents of mine.



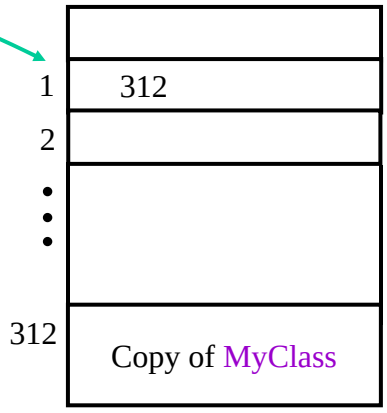
Remark 2: We see that the declared symbol is merely a pointer (reference) to the object.

36

APIs and class use

Memory diagram: Object declaration and construction

- We have
  MyClass mine = new MyClass();

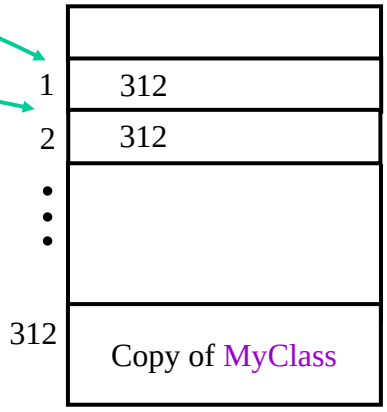


37

APIs and class use

Memory diagram: Object declaration and construction

- We have
  MyClass mine = new MyClass();
- Suppose we add
  MyClass yours = mine;
- From now on, if we change the object referred to by mine, then the same changes occur to yours, e.g.,
  - If mine is a Stock object with a price and the price of mine increases; so does the price of yours.



38

APIs and class use

The null reference

- An object variable contains a reference to an object.
- Unlike variables of the primitive types, an object variable can refer to no particular object.
- We use the Java keyword null to accomplish this.
- Note, however, that attempts to invoke a method on a null object reference terminates the offending program.

39

APIs and class use

The null reference

- Example

  String greeting = “Hello”;

  String message = “”; // the empty string

  String comment = null; // refers to no string at all

40

APIs and class use

The null reference

- Example

```
String greeting = “Hello”;
String message = “”; // the empty string
String comment = null; // refers to no string at all
int g = greeting.length( ); // 5
int m = message.length( ); // 0
int c = comment.length( ); // program terminates
```

41

APIs and class use

Using instance methods

- To invoke an instance (non-static) method *m* in a class *C*, the following steps are followed:
  1. Create an instance *x* of *C*: *C x = new C(parameters)*
  2. Determine the signature of *m*: parameters, their data type and their order.
  3. Invoke using the dot: *x.m(parameters)*
- Example:

```
Stock s = new Stock(“BMO”);
double tradingPrice = s.getPrice();
```

APIs and class use

Using instance methods

- To invoke an instance (non-static) method *m* in a class *C*, the following steps are followed:
  1. Create an instance *x* of *C*: *C x = new C(parameters)*
  2. Determine the signature of *m*: parameters, their data type and their order.
  3. Invoke using the dot: *x.m(parameters)*
- Example: Stock s = new Stock("BMO");
  double tradingPrice = s.getPrice();

Remarks

- A void method, e.g., setSymbol() is often in a statement by itself.
- A non-void method getPrice() is usually part of a statement.

43

APIs and class use

Using static methods in a non-static class

- To access a static method *m* in a (non-static) class *C*, the following steps can be followed:
  1. Ensure that class *C* is reachable (already available or imported).
  2. Access using the dot: *C.m*
- Alternatively,
  1. Create an instance object, *o*, of the class *C*.
  2. Access using the dot: *o.m*

44

APIs and class use

Using static methods in a non-static class

- Example: There could be (but there is not) a static method in the Stock class

```
getExchangeCompIndexValue();
```

- One could invoke such a method either through

```
import type.lib.*;
output.println(Stock.getExchangeCompIndexValue());
```

or through

```
import type.lib.*;
Stock myStock = new Stock("BMO");
output.println(myStock.getExchangeCompIndexValue());
```

45

APIs and class use

Using instance attributes

- To access an instance (non-static) attribute *a* in a class *C*, the following steps can be followed:
  1. Ensure that an instance object *o* of the class *C* has been created.
  2. Access using the dot: *o.a*
- Example:

```
Stock myStk = new Stock("BMO");
output.println(myStk.name);
```

46

APIs and class use

Remarks

- It is preferable to access instance attributes through methods designed for this purpose.
- Example

```
output.println(myStk.getName());
```

- Indeed, good class design typically only allows access to instance attributes through methods.
  - The attributes will be encapsulated and not even appear in the API.
- This practice prevents users from operations that can lead to inconsistency within an object.
- Example

```
// bad practice follows
mkStk.name = “Inconsistent with other fields”;
```

47

APIs and class use

Using static attributes in a non-static class

- To access a static attribute *a* in a (non-static) class *C*, the following steps can be followed:
  1. Ensure that class *C* is reachable (already available or imported).
  2. Access using the dot: *C.a*
- Alternatively,
  1. Create an instance object, *o*, of the class *C*.
  2. Access using the dot: *o.a*

48

APIs and class use

Using static attributes in a non-static class

- Example

```
import type.lib.*;
output.println(Stock.titleCaseName); // false
```

49

APIs and class use

Using static attributes in a non-static class

- Example

```
import type.lib.*;
output.println(Stock.titleCaseName); // false
Stock.titleCaseName = true;
output.println(Stock.titleCaseName); // true
```

APIs and class use

Using static attributes in a non-static class

- Example

```
import type.lib.*;
output.println(Stock.titleCaseName); // false
Stock.titleCaseName = true;
output.println(Stock.titleCaseName); // true
Stock stk1 = new Stock("HR.X");
Stock stk2 = new Stock("HR.Y");
```

APIs and class use

Using static attributes in a non-static class

- Example

```
import type.lib.*;
output.println(Stock.titleCaseName); // false
Stock.titleCaseName = true;
output.println(Stock.titleCaseName); // true
Stock stk1 = new Stock("HR.X");
Stock stk2 = new Stock("HR.Y");
output.println(stk1.getName()); // printed in title case
```

APIs and class use

Using static attributes in a non-static class

- Example

```
import type.lib.*;
output.println(Stock.titleCaseName); // false
Stock.titleCaseName = true;
output.println(Stock.titleCaseName); // true
Stock stk1 = new Stock("HR.X");
Stock stk2 = new Stock("HR.Y");
output.println(stk1.getName()); // printed in title case
stk2.titleCaseName = false;
```

APIs and class use

Using static attributes in a non-static class

- Example

```
import type.lib.*;
output.println(Stock.titleCaseName); // false
Stock.titleCaseName = true;
output.println(Stock.titleCaseName); // true
Stock stk1 = new Stock("HR.X");
Stock stk2 = new Stock("HR.Y");
output.println(stk1.getName()); // printed in title case
stk2.titleCaseName = false;
output.println(stk1.getName()); // printed in ???
```

APIs and class use

Using static attributes in a non-static class

- Example

```
import type.lib.*;
output.println(Stock.titleCaseName); // false
Stock.titleCaseName = true;
output.println(Stock.titleCaseName); // true
Stock stk1 = new Stock("HR.X");
Stock stk2 = new Stock("HR.Y");
output.println(stk1.getName()); // printed in title case
stk2.titleCaseName = false;
output.println(stk1.getName()); // printed in all caps
```

APIs and class use

Using static attributes in a non-static class

- Example

```
import type.lib.*;
output.println(Stock.titleCaseName); // false
Stock.titleCaseName = true;
output.println(Stock.titleCaseName); // true
Stock stk1 = new Stock("HR.X");
Stock stk2 = new Stock("HR.Y");
output.println(stk1.getName()); // printed in title case
stk2.titleCaseName = false;
output.println(stk1.getName()); // printed in all caps
Stock.titleCaseName = true;
```

APIs and class use

Using static attributes in a non-static class

- Example

```
import type.lib.*;
output.println(Stock.titleCaseName); // false
Stock.titleCaseName = true;
output.println(Stock.titleCaseName); // true
Stock stk1 = new Stock("HR.X");
Stock stk2 = new Stock("HR.Y");
output.println(stk1.getName()); // printed in title case
stk2.titleCaseName = false;
output.println(stk1.getName()); // printed in all caps
Stock.titleCaseName = true;
output.println(stk2.getName()); // printed in title case
```

57

APIs and class use

The toString() method

- Here we have a standard method that returns a string representing the object upon which it is invoked.
- Example

```
Stock stk1 = new Stock("BMO");
 // print BMO Bank of Montreal
output.println(stk1.toString());
```

58

APIs and class use

The toString() method

- Here we have a standard method that returns a string representing the object upon which it is invoked.
- Example

```
Stock stk1 = new Stock("BMO");
 // print BMO Bank of Montreal
output.println(stk1.toString());
```

- Indeed, output methods automatically invoke the toString method when outputting object references.
- Example, the following fragments yield the same output

```
output.println(stk1.toString());
output.println(stk1);
```

59

APIs and class use

The toString() method

- Getting an intuitive return from toString depends on the class to which it is being applied having appropriately defined toString for that particular class.
- For example, we have seen good design in this regard for the Stock class.

60

APIs and class use

The toString() method

- Getting an intuitive return from toString depends on the class to which it is being applied having appropriately defined toString for that particular class.
- For example, we have seen good design in this regard for the Stock class.
- If the designer of the class has not done this task, then Java makes use of a default toString operation.
  - Returns the the name and address of the object (in hex).
- Example

```
StockNS myStk = new StockNS("BMO");
// prints something like type.lib.StockBMO@86d4c1
output.println(myStk.toString());
```

61

APIs and class use

Comparing objects

- You *cannot* use the relational operators to compare two objects.
- For example, given two objects o1 and o2, in the statement

```
boolean test = o1 == o2;
```

test is true if the object references are the same, not whether the objects are the same.

62

APIs and class use

Comparing objects

- You *cannot* use the relational operators to compare two objects.
- For example, given two objects o1 and o2, in the statement

  ```
  boolean test = o1 == o2;
  ```

  test is true if the object references are the same, not whether the objects are the same.
- As an example with the Stock class

  ```
  Stock s1 = new Stock("BMO");
  Stock s2 = new Stock("BMO");
  Stock s3 = s2;
  boolean compare2and3 = s2 == s3; // true
  boolean compare1and2 = s1 == s2; // false
  ```

63

APIs and class use

Comparing objects

- Use of == is only appropriate for comparing object references.
- Java is set up so that for any two objects o1 and o2, o1.equals(o2) is always defined.
- However, for arbitrary objects Java has no idea what it means to perform a comparison
  - It defaults to using ==
  - This can have unintended effects in the hands of a naïve user!

64

APIs and class use

Comparing objects

- The designer of a class can (*but need not*) override the default and sensibly define equals(o) for a given class
  - e.g., as has been done for Stock

```
Stock s1 = new Stock("BMO");
Stock s2 = new Stock("BMO");
boolean compare1and2 = s1 == s2; // false
compare1and2 = s1.equals(s2); // true
```

- Always check to see if equals() is appropriately defined for a class of objects before employing it
  - e.g., by consulting the class API

65

APIs and class use

Comparing objects

Method Summary

boolean	equals(java.lang.Object other) Test the equality of stock objects.

66

APIs and class use

Comparing objects

Method Detail
equals

public boolean equals(java.lang.Object other)

Test the equality of stock objects. An object is considered equal to this one if it is a stock object with the same symbol as this one.

Overrides:
equals in class java.lang.Object
Parameters:
other - a reference to the object to test equality with.
Returns:
true if other is not null and it points to an object that is equal (as defined above) to this object. The return is false otherwise.

67

APIs and class use

Testing for the null object reference

- It is appropriate to use == when we want to test for the null object reference.

```
Stock s = null;
boolean test = s == null; // test is true
```

- This make sense, since == compares object references (as opposed to the objects themselves).

68

APIs and class use

Checking for object class

- In some situations it is desirable to make sure that an object belongs to (is an instance of) a particular class.

```
Stock myStock;
boolean test = (myStock instanceof Stock); // error
myStock = new Stock (“BMO”);
test = (myStock instanceof Stock); // true
myStock = null;
test = ( myStock instanceof Stock); // false
```

- For example, such tests can help safeguard against inappropriate actions on an object.

69

APIs and class use

Cloning

- As a way to create copies of an object, the Stock class provides an instance method cloneMe().
- This method works by copying all the fields of the object on which it is invoked.
- Example

```
Stock s1 = new Stock(“TD”);
s1.setDelimiter(‘$’);
Stock s2 = new Stock(s1.getSymbol());
Stock s3 = s1.cloneMe();
```

70

APIs and class use

Cloning

- As a way to create copies of an object, the Stock class provides an instance method cloneMe().
- This method works by copying all the fields of the object on which it is invoked.
- Example

```
Stock s1 = new Stock("TD");
s1.setDelimiter('$');
Stock s2 = new Stock(s1.getSymbol());
Stock s3 = s1.cloneMe();
output.println(s1 == s2); // false
output.println(s1 == s3); // false
```

71

APIs and class use

Cloning

- As a way to create copies of an object, the Stock class provides an instance method cloneMe().
- This method works by copying all the fields of the object on which it is invoked.
- Example

```
Stock s1 = new Stock("TD");
s1.setDelimiter('$');
Stock s2 = new Stock(s1.getSymbol());
Stock s3 = s1.cloneMe();
output.println(s1 == s2); // false
output.println(s1 == s3); // false
output.println(s1.equals(s2)); // true
output.println(s1.equals(s3)); // true
```

72

APIs and class use

Cloning

- As a way to create copies of an object, the Stock class provides an instance method cloneMe().
- This method works by copying all the fields of the object on which it is invoked.
- Example

```
Stock s1 = new Stock("TD");
s1.setDelimiter('$');
Stock s2 = new Stock(s1.getSymbol());
Stock s3 = s1.cloneMe();
output.println(s1 == s2); // false
output.println(s1 == s3); // false
output.println(s1.equals(s2)); // true
output.println(s1.equals(s3)); // true
output.println(s1); // TD$Toronto-Dominion Bank
output.println(s2); // TD Toronto-Dominion Bank
output.println(s3); // TD$Toronto-Dominion Bank
```

73

APIs and class use

Cloning

- As a way to create copies of an object, the Stock class provides an instance method cloneMe().
- This method works by copying all the fields of the object on which it is invoked.

Remark

- Do not confuse the cloneMe method (as defined in the Stock class) with the Java clone method.
- At this point, we are not concerned with Java clone.

74

Outline

- Objects
- API and usage
- **Patterns**
- (More) usage
- Abstraction: Objects, classes & methods
- Software engineering example

75

Patterns

A class can have

- constructors
  - Used to create instances (objects) that belong to the class,
  - e.g., the Stock constructors.

Patterns

A class can have

- constructors
  - Used to create instances (objects) that belong to the class,
  - e.g., the Stock constructors.
- instance methods (non-static)
  - Operators that belong to and can be invoked on each object that is an instance of the class,
  - e.g., myStk.getPrice().

Patterns

A class can have

- constructors
  - Used to create instances (objects) that belong to the class,
  - e.g., the Stock constructors.
- instance methods (non-static)
  - Operators that belong to and can be invoked on each object that is an instance of the class,
  - e.g., myStk.getPrice().
- instance attributes (non-static)
  - Stores for data associated with the object and which can be retrieved and changed (unless final) by users
  - e.g., myStk.name.

78

Patterns

A class can have

79

Patterns

A class can have

- static methods
  - That belong to the class as a whole,
  - e.g., the hypothetical Stock.getExchangeCompIndexValue().

80

Patterns

A class can have

- static methods
  - That belong to the class as a whole,
  - e.g., the hypothetical Stock.getExchangeCompIndexValue().
- static attributes
  - Stores for data associated with the class as a whole and which can be retrieved and changed (unless final) by users
  - e.g., Stock.titleCaseName.

81

Patterns

Non-static classes can have	Static classes can have
constructors	
instance (non-static) methods	
instance (non-static) attributes	
static methods	static methods
static attributes	static attributes

82

Patterns

Types of methods

- accessor methods
  - Used to retrieve the value of an attribute of the object,
  - e.g., in Stock we have getSymbol, getName, getDelimiter, etc.
- mutator methods
  - Used to change the value of an attribute of an object
  - e.g., in Stock we have setSymbol, setDelimiter, etc.

83

Patterns

Types of methods

- standard methods
  - Defined for all classes (even if not by the class designer),
  - e.g., toString, equals, etc.
- specialized methods
  - Particular to a class,
  - e.g., in Stock we have refresh.

84

Patterns

Visibility

- All of the constructors, methods and attributes that are available to the user of a class are said to be public or visible.
- All such features appear in the API for the class so you know they are there for you to exploit.
- We notice that in the API detail these features are marked with the Java keyword public.
- There also may be private features, but these are not available to the user and are not visible in the API.

85

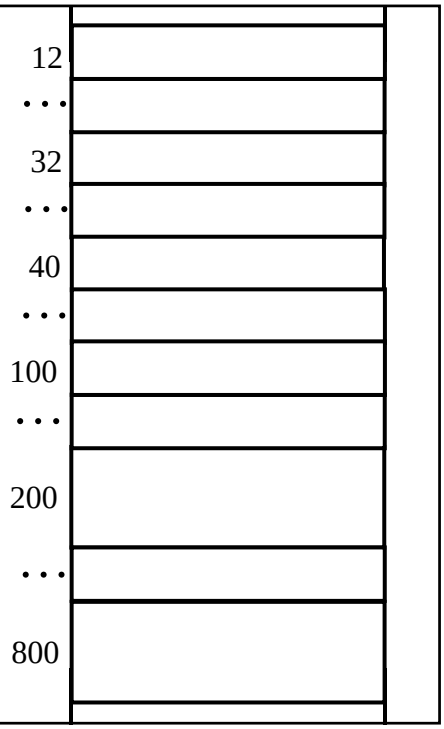
Outline

- Objects
- API and usage
- Patterns
- **(More) usage**
- Abstraction: Objects, classes & methods
- Software engineering example

86

(More) class usage

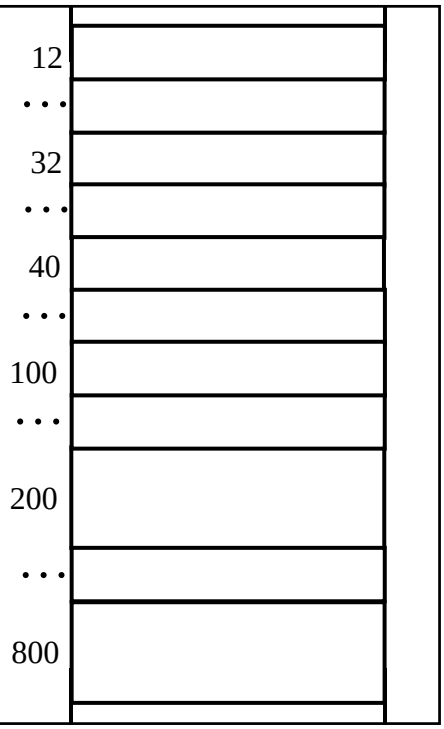
Object vs. object reference



(More) class usage

Object vs. object reference

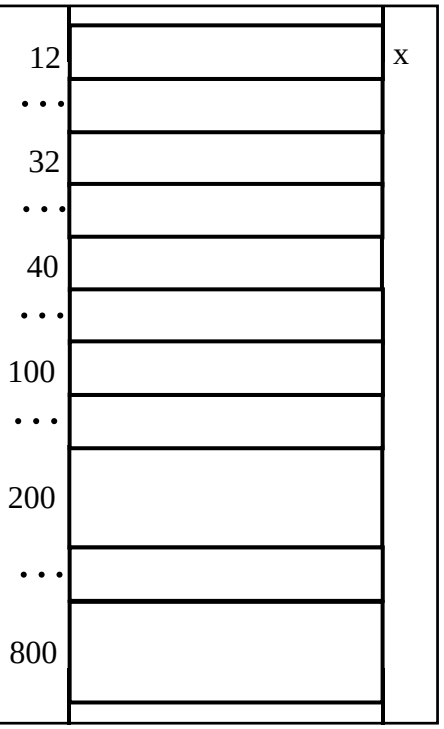
```
import type.lib.*;
```



(More) class usage

Object vs. object reference

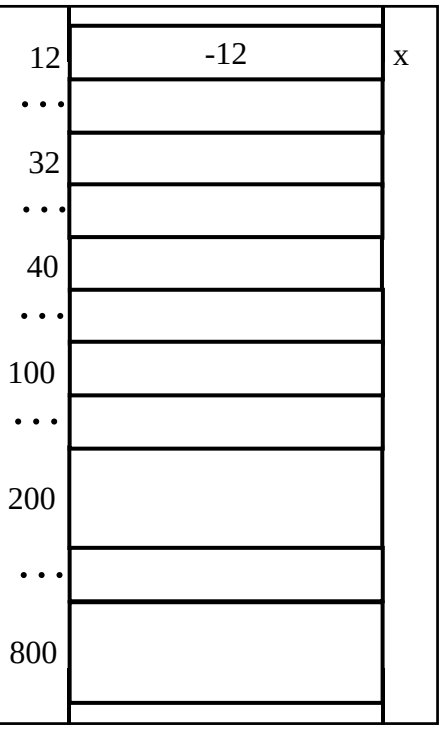
```
import type.lib.*;
int x;
```



(More) class usage

Object vs. object reference

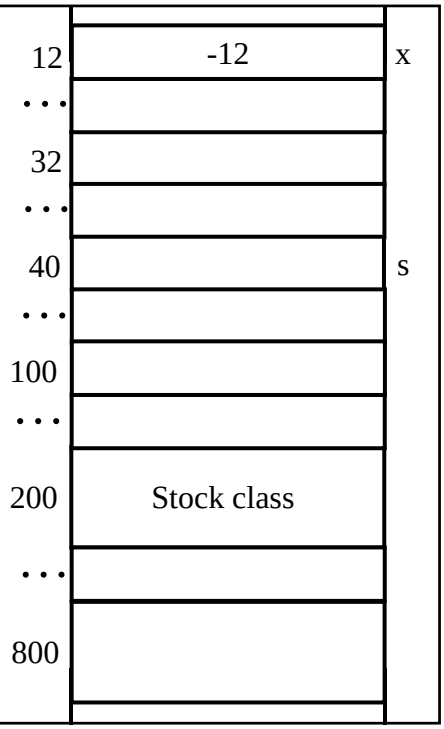
```
import type.lib.*;
int x;
x = -12;
```



(More) class usage

Object vs. object reference

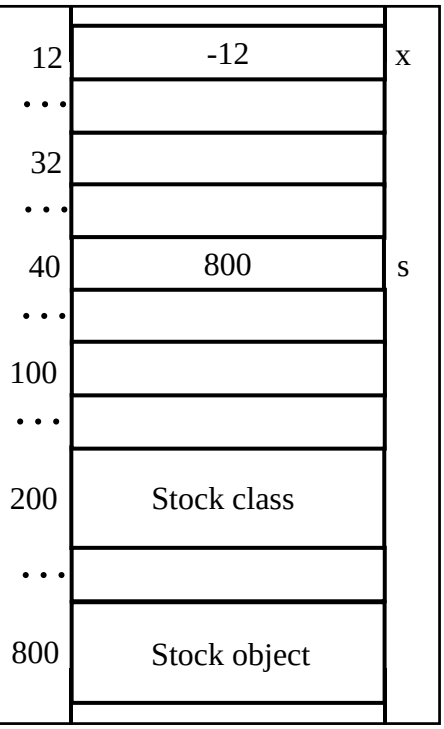
```
import type.lib.*;
int x;
x = -12;
Stock s;
```



(More) class usage

Object vs. object reference

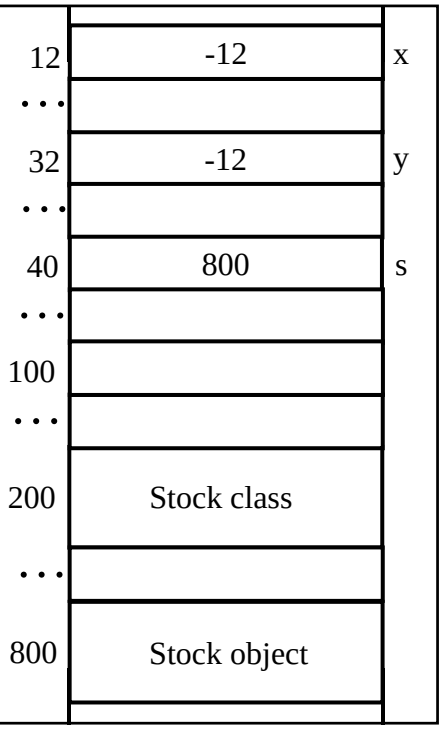
```
import type.lib.*;
int x;
x = -12;
Stock s;
s = new Stock("RY");
```



(More) class usage

Object vs. object reference

```
import type.lib.*;
int x;
x = -12;
Stock s;
s = new Stock("RY");
int y = x;
assert x == y; // assert okay
```

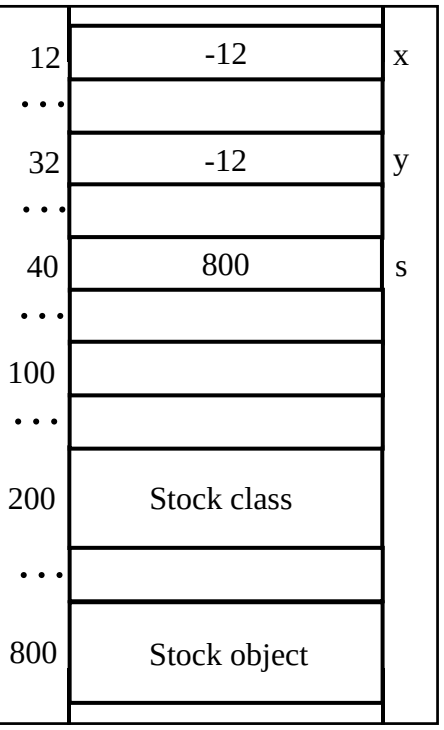


(More) class usage

Object vs. object reference

```
import type.lib.*;
int x;
x = -12;
Stock s;
s = new Stock("RY");
int y = x;
assert x == y; // assert okay
```

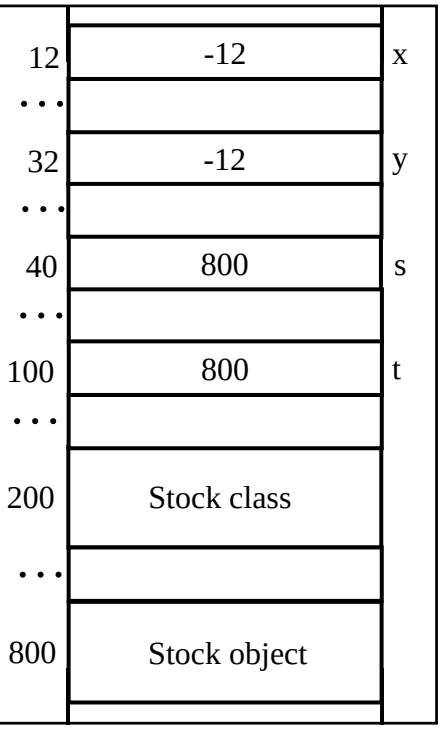
Don't forget: To enable assertions use
% java –ea app



(More) class usage

Object vs. object reference

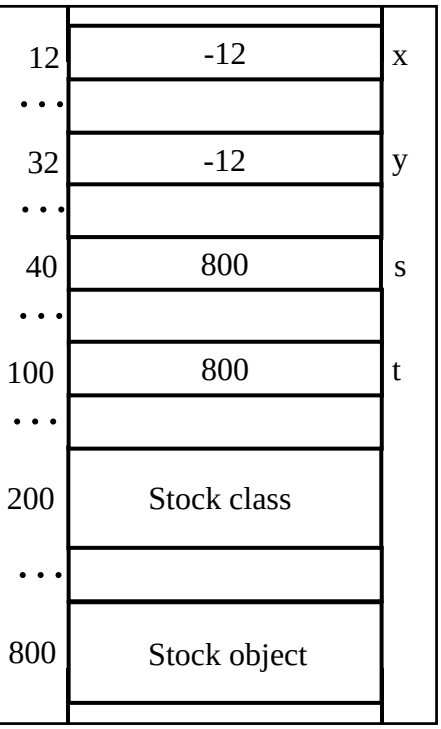
```
import type.lib.*;
int x;
x = -12;
Stock s;
s = new Stock("RY");
int y = x;
assert x == y; // assert okay
Stock t = s;
assert s ==  t;
```



(More) class usage

Object vs. object reference

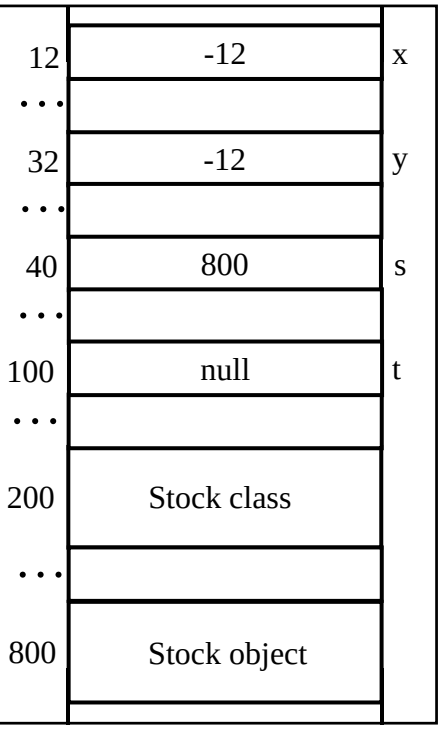
```
import type.lib.*;
int x;
x = -12;
Stock s;
s = new Stock("RY");
int y = x;
assert x == y; // assert okay
Stock t = s;
assert s ==  t; // assert okay
```



(More) class usage

Object vs. object reference

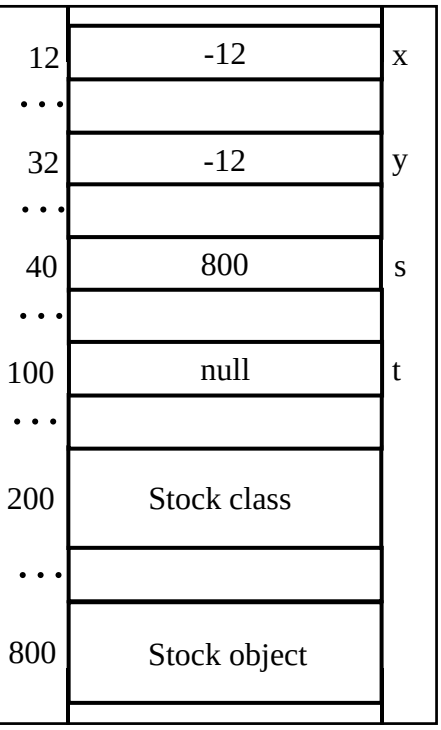
```
import type.lib.*;
int x;
x = -12;
Stock s;
s = new Stock("RY");
int y = x;
assert x == y; // assert okay
Stock t = s;
assert s ==  t; // assert okay
t = null;
assert s == t;
```



(More) class usage

Object vs. object reference

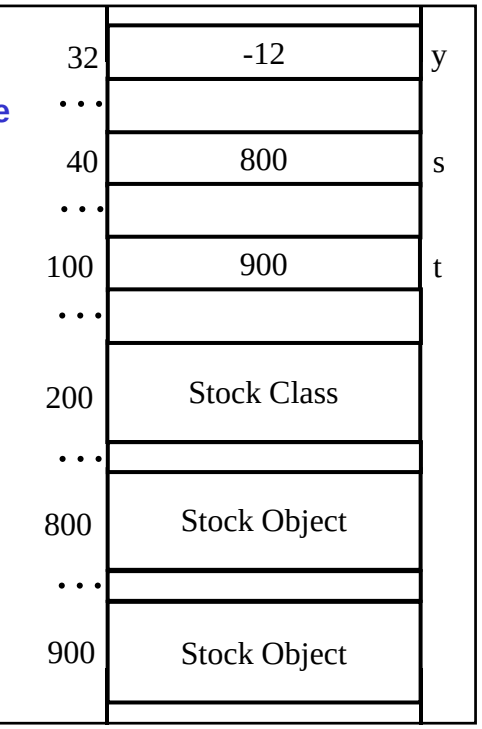
```
import type.lib.*;
int x;
x = -12;
Stock s;
s = new Stock("RY");
int y = x;
assert x == y; // assert okay
Stock t = s;
assert s ==  t; // assert okay
t = null;
assert s == t; // assert fails
```



(More) class usage

Object vs. object reference

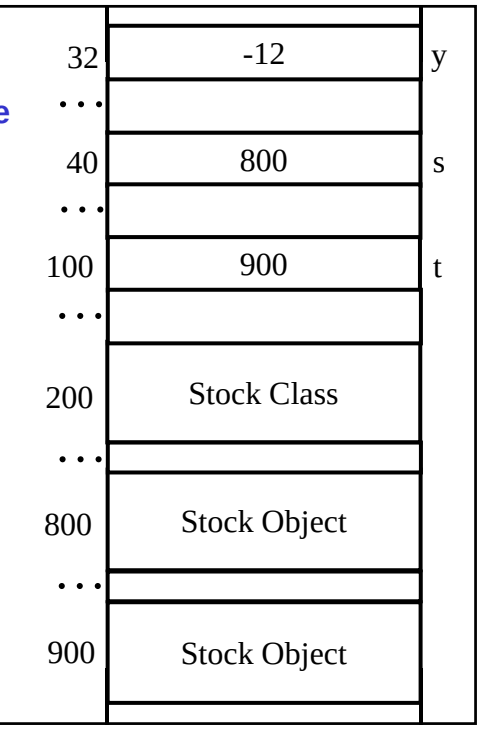
```
import type.lib.*;
int x;
x = -12;
Stock s;
s = new Stock("RY");
int y = x;
assert x == y; // assert okay
Stock t = s;
assert s ==  t; // assert okay
t = null;
assert s == t; // assert fails
t = new Stock("RY");
assert s == t;
```



(More) class usage

Object vs. object reference

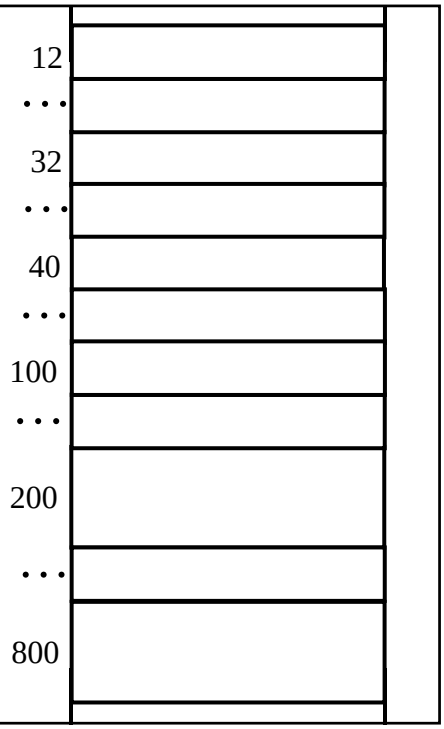
```
import type.lib.*;
int x;
x = -12;
Stock s;
s = new Stock("RY");
int y = x;
assert x == y; // assert okay
Stock t = s;
assert s ==  t; // assert okay
t = null;
assert s == t; // assert fails
t = new Stock("RY");
assert s == t; // assert fails
```

(More) class usage

Customization

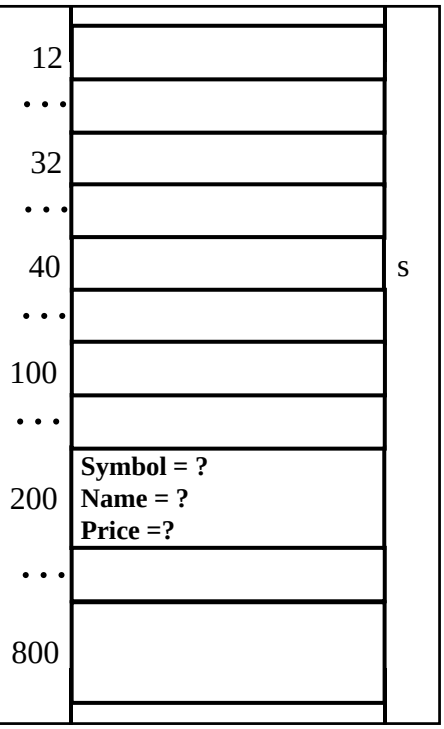
- Upon importing
  import type.lib.*;
  we have: memory diagram



(More) class usage

Customization

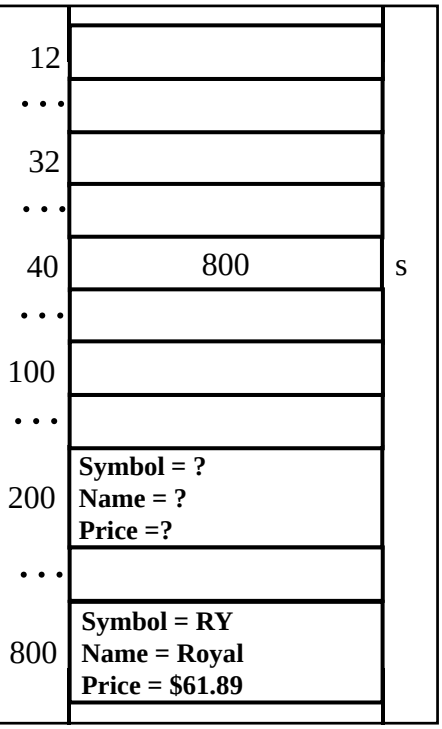
- Upon importing
  import type.lib.*;
  we have: memory diagram
- Upon declaration
  Stock s;
  we have: memory diagram



(More) class usage

Customization

- Upon importing
  import type.lib.*;
  we have: memory diagram
- Upon declaration
  Stock s;
  we have: memory diagram
- Upon creation
  s = new Stock(“RY);
  we have: memory diagram

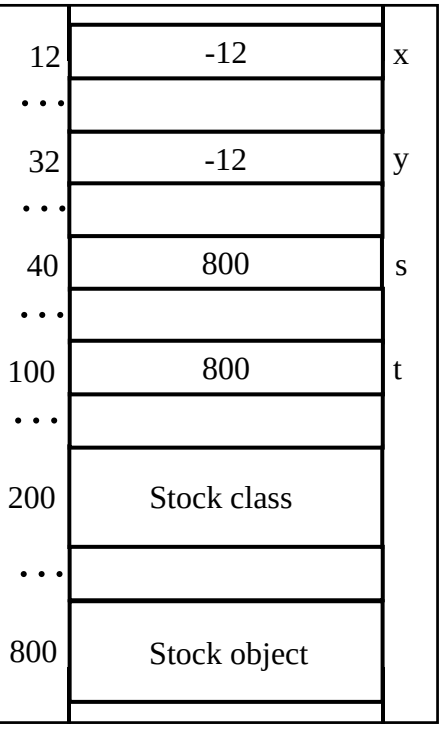


(More) class usage

Method toString()

IO.println(x);
IO.println(s);
IO.println(t);
IO.println(s.toString());
IO.println(t.toString());

- Upon seeing println(s) Java asks if toString() is defined for s.
- If so, then it invokes s.toString() for the value to be printed.
- Otherwise it prints the value of the reference (e.g., 800).



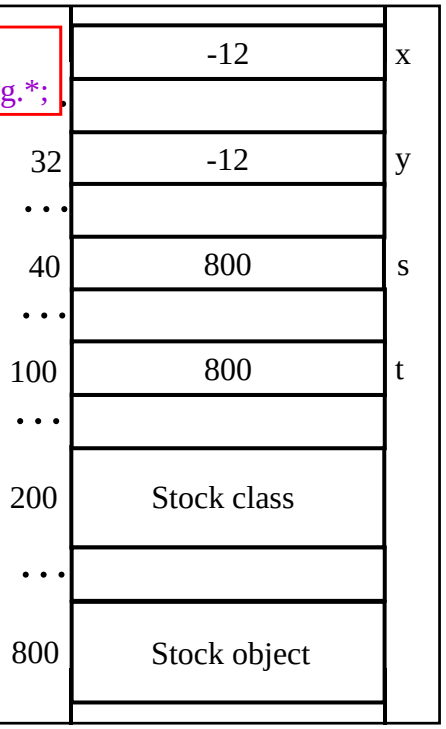
(More) clas

To gain access
import type.lang.*;

Method toString()

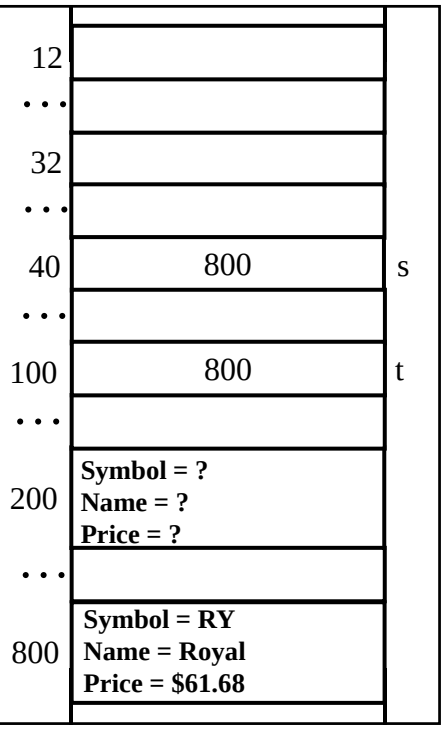
```
IO.println(x);
IO.println(s);
IO.println(t);
IO.println(s.toString());
IO.println(t.toString());
```

- Upon seeing println(s) Java asks if toString() is defined for s.
- If so, then it invokes s.toString() for the value to be printed.
- Otherwise it prints the value of the reference (e.g., 800).



(More) class usage

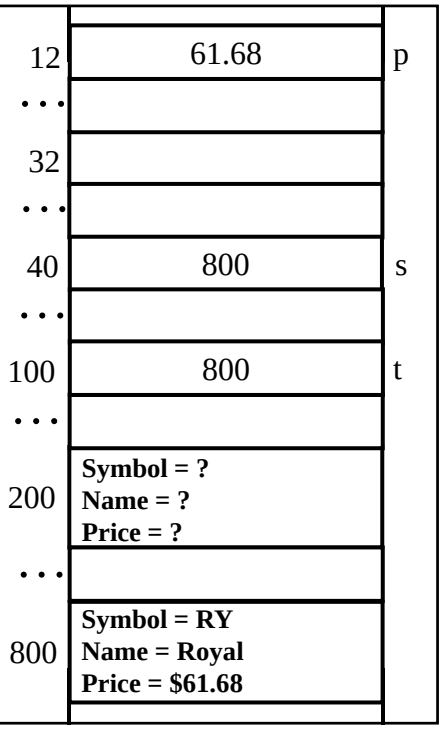
Accessing & setting field values



(More) class usage

Accessing & setting field values

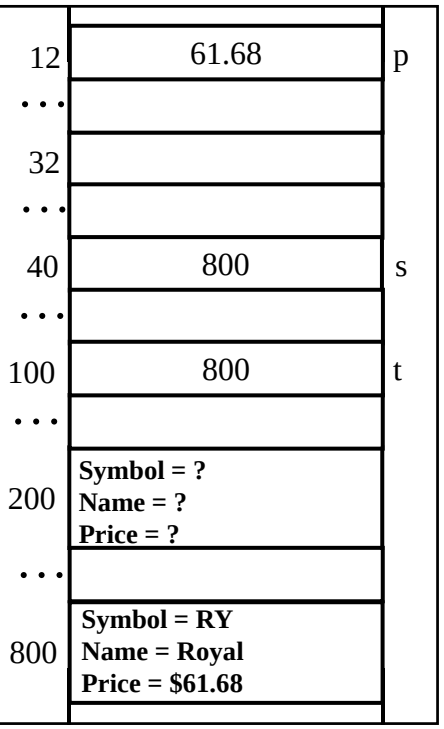
```
double p = s.getPrice();
// next 2 lines yield same result
IO.println(p);
IO.println(t.getPrice());
```



(More) class usage

Accessing & setting field values

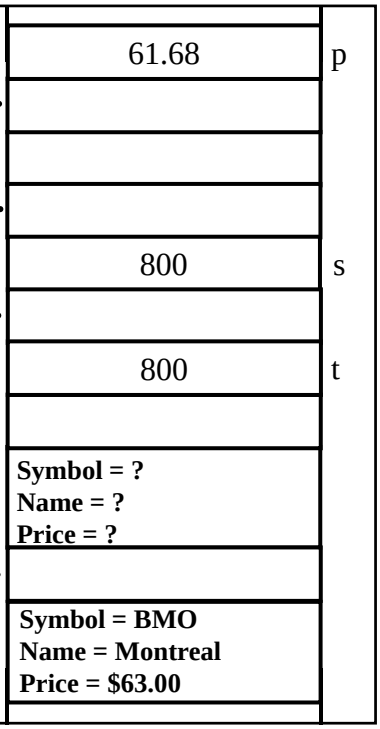
```
double p = s.getPrice();
// next 2 lines yield same result
IO.println(p);
IO.println(t.getPrice());
// but not the next 2
IO.println(s.getName());
IO.println(s.getSymbol());
```



(More) class usage

Accessing & setting field values

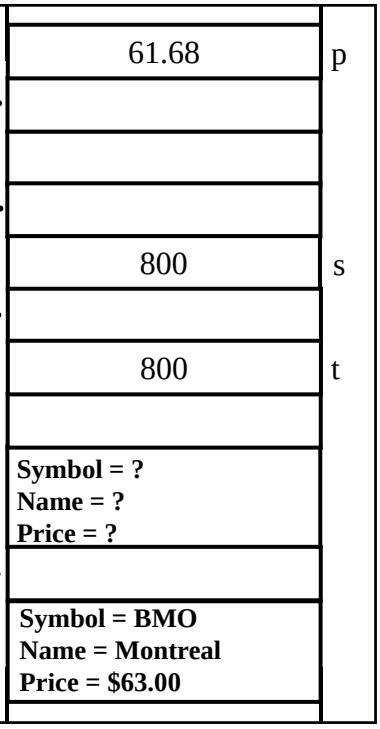
```
double p = s.getPrice();
// next 2 lines yield same result
IO.println(p);
IO.println(t.getPrice());
// but not the next 2
IO.println(s.getName());
IO.println(s.getSymbol());
s.setSymbol("BMO");
// what about these 2
IO.println(s);
IO.println(t);
```



(More) class usage

Accessing & setting field values

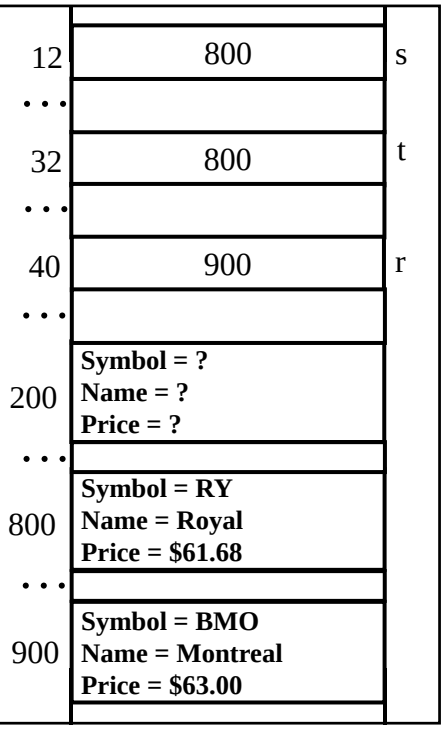
```
double p = s.getPrice();
// next 2 lines yield same result
IO.println(p);
IO.println(t.getPrice());
// but not the next 2
IO.println(s.getName());
IO.println(s.getSymbol());
s.setSymbol("BMO");
// what about these 2: also same
IO.println(s);
IO.println(t);
```



(More) class usage

== vs. equals()

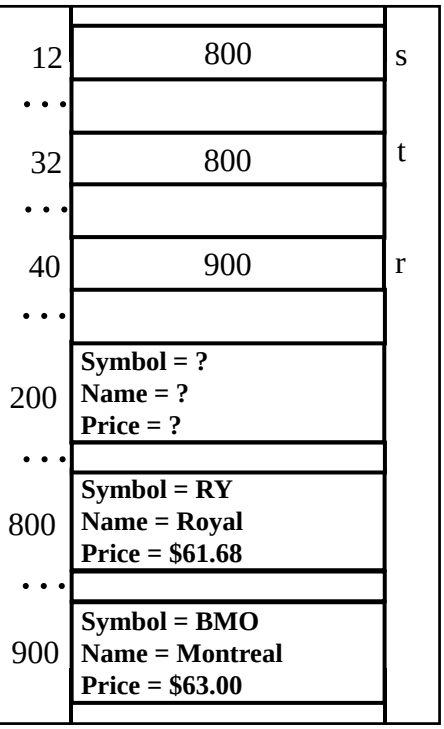
```
Stock s, t, r;
s = new Stock("RY");
t =s;
r = new Stock("BMO");
```



(More) class usage

== vs. equals()

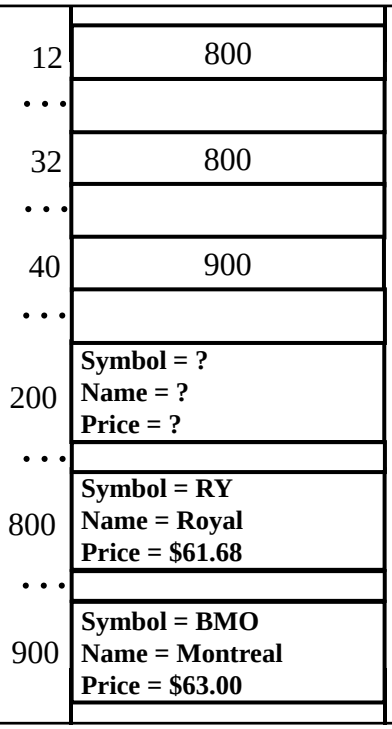
```
Stock s, t, r;
s = new Stock("RY");
t =s;
r = new Stock("BMO");
assert s == t;
assert s != r;
```



(More) class usage

== vs. equals()

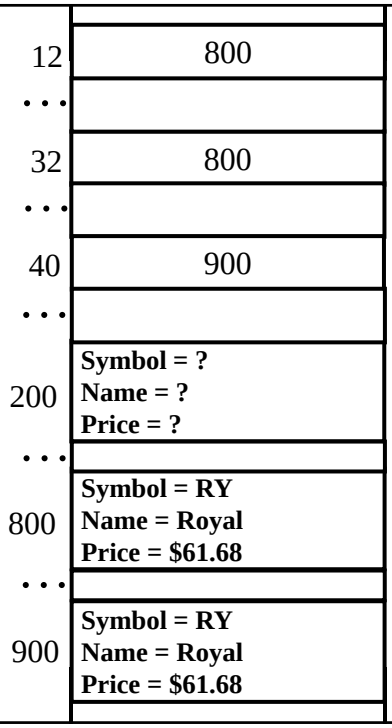
```
Stock s, t, r;
s = new Stock("RY");
t =s;
r = new Stock("BMO");
assert s == t; // okay
assert s != r; // okay
```



(More) class usage

== vs. equals()

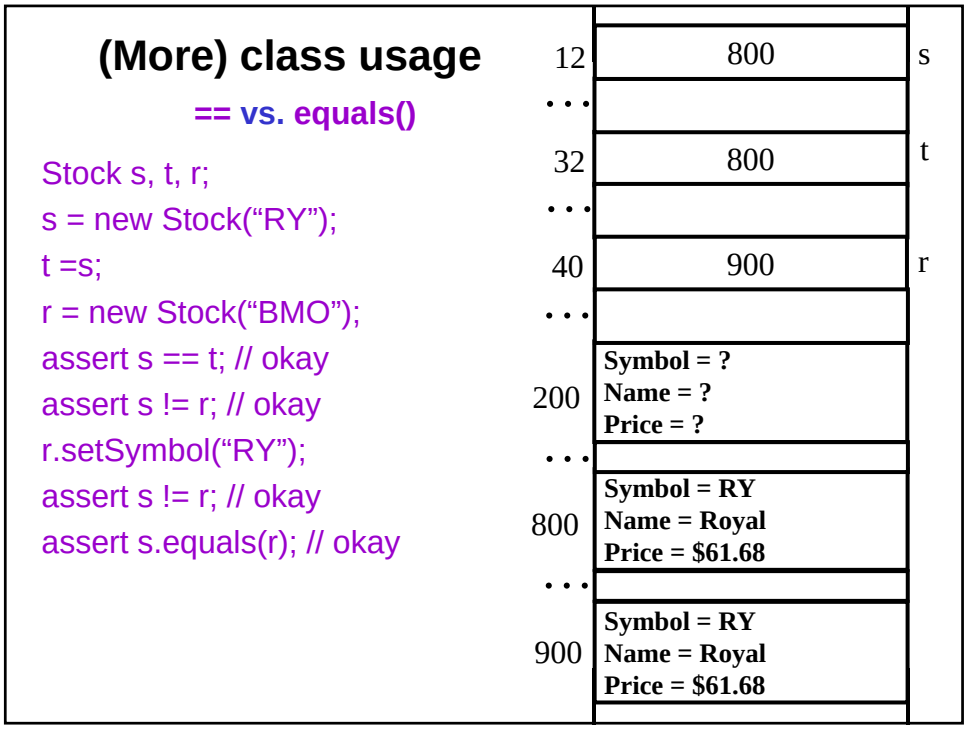
```
Stock s, t, r;
s = new Stock("RY");
t =s;
r = new Stock("BMO");
assert s == t; // okay
assert s != r; // okay
r.setSymbol("RY");
assert s != r;
assert s.equals (r);
```



(More) class usage

== vs. equals()

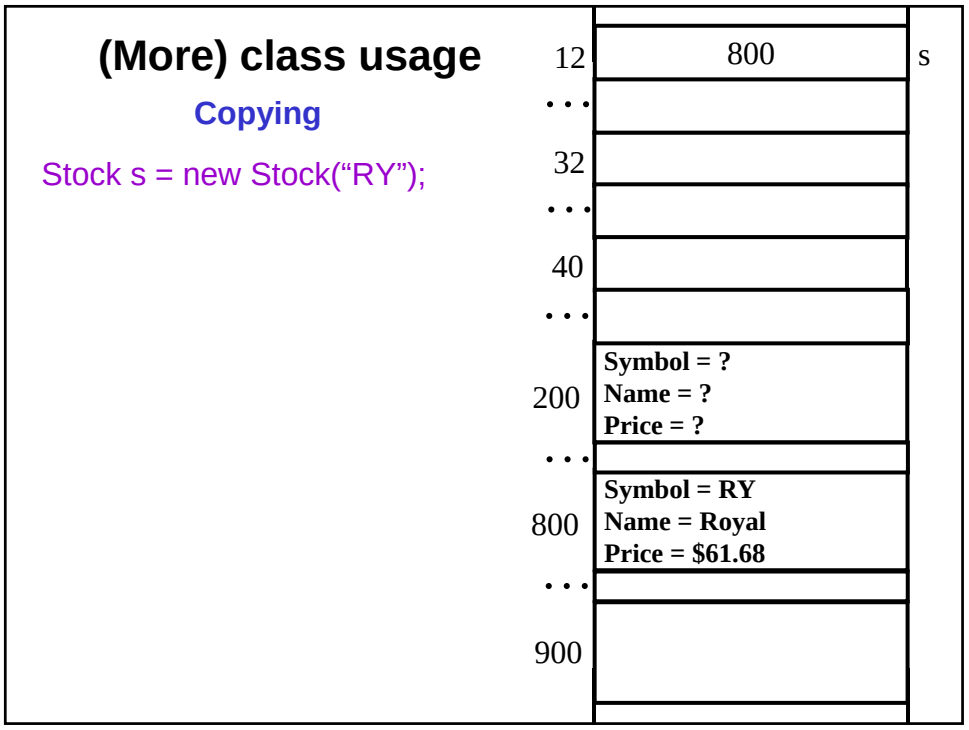
```
Stock s, t, r;
s = new Stock("RY");
t =s;
r = new Stock("BMO");
assert s == t; // okay
assert s != r; // okay
r.setSymbol("RY");
assert s != r; // okay
assert s.equals(r); // okay
```



(More) class usage

Copying

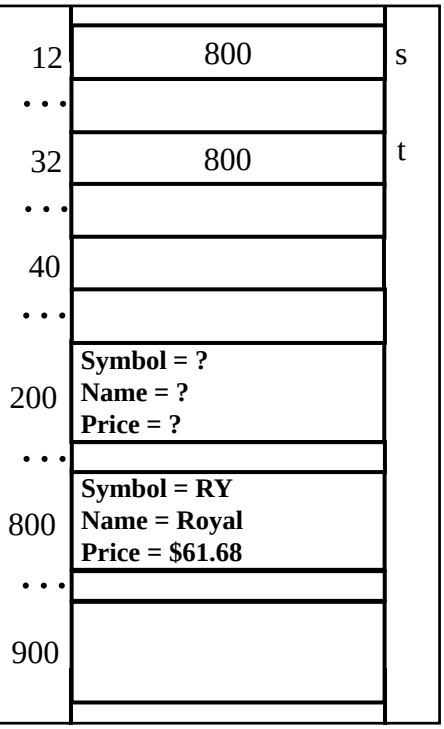
```
Stock s = new Stock("RY");
```



(More) class usage

Copying

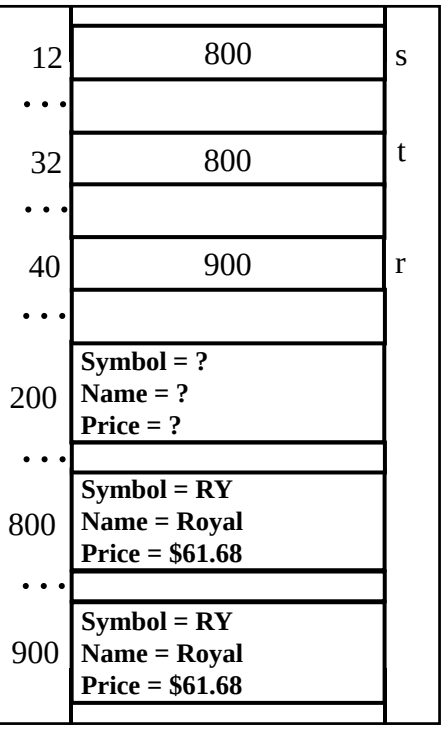
Stock s = new Stock("RY");
Stock t = s;



(More) class usage

Copying

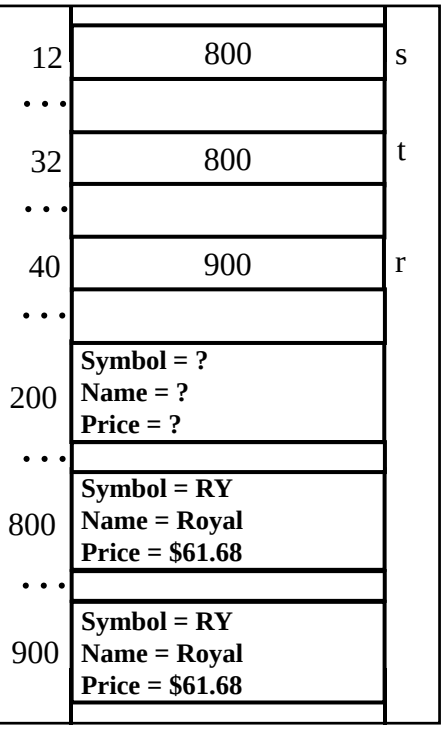
Stock s = new Stock("RY");
Stock t = s;
Stock r = s.cloneMe();



(More) class usage

Copying

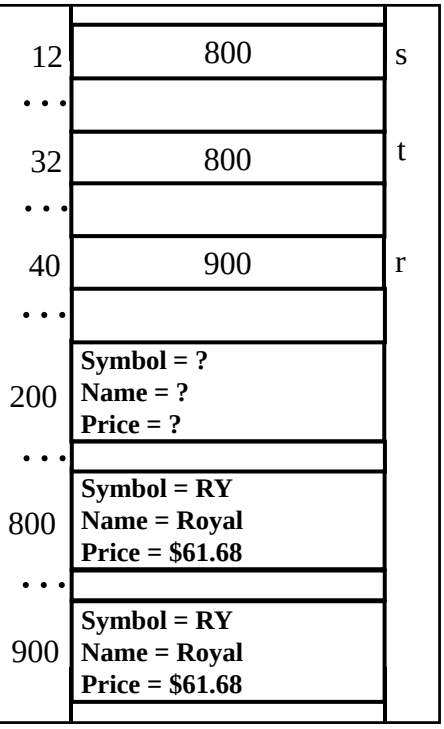
```
Stock s = new Stock("RY");
Stock t = s;
Stock r = s.cloneMe();
boolean case1 = s==t;
boolean case2 = s==r;
case1 = s.equals(t);
case2 = s.equals(r);
```



(More) class usage

Copying

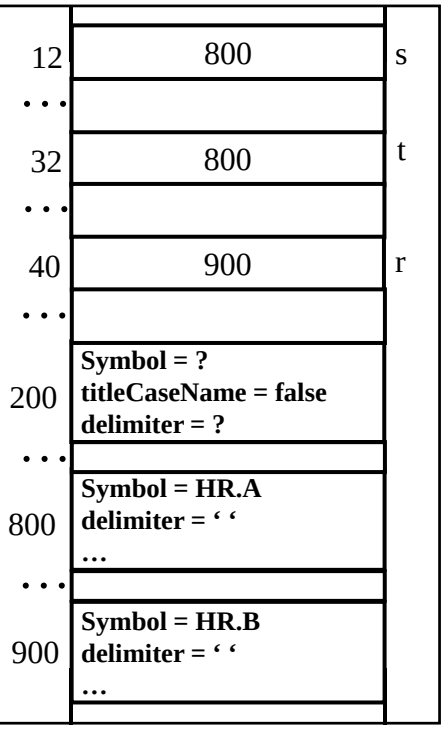
```
Stock s = new Stock("RY");
Stock t = s;
Stock r = s.cloneMe();
boolean case1 = s==t; // true
boolean case2 = s==r; // false
case1 = s.equals(t); // true
case2 = s.equals(r); // true
```



(More) class usage

static vs. non-static

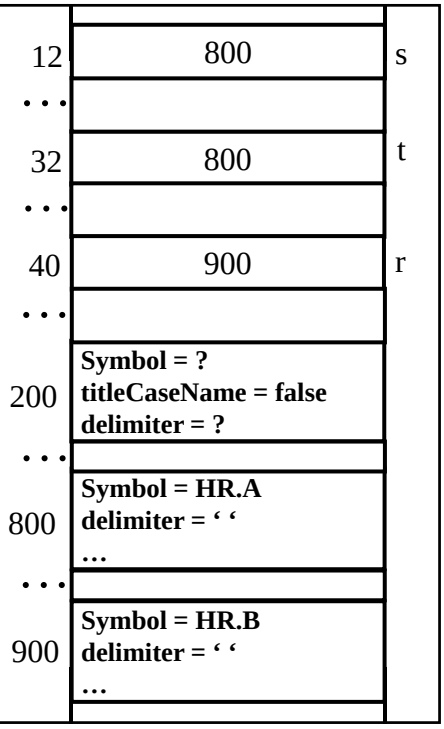
```
Stock s, t, r;
s = new Stock("HR.A");
t = s;
r = new Stock("HR.B");
```



(More) class usage

static vs. non-static

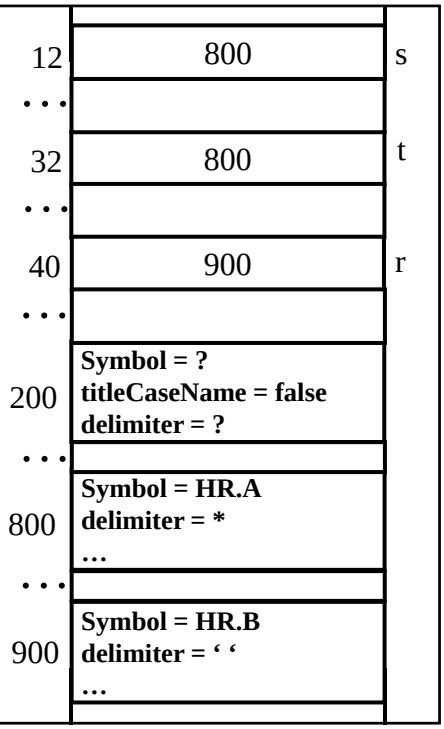
```
Stock s, t, r;
s = new Stock("HR.A");
t = s;
r = new Stock("HR.B");
IO.println(s); //HR.A ACOMP CORP
```



(More) class usage

static vs. non-static

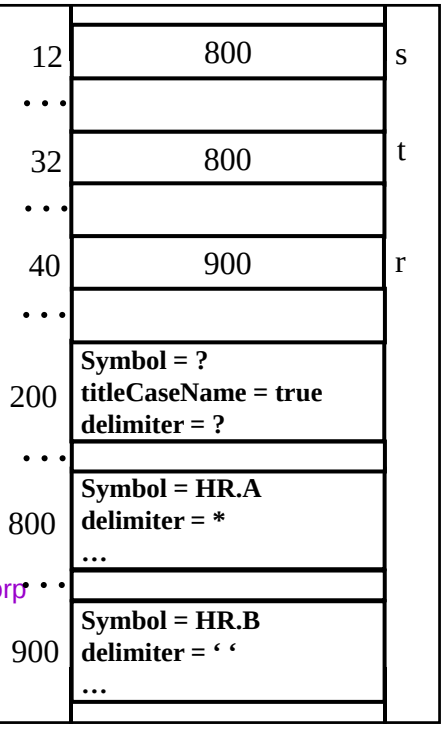
```
Stock s, t, r;
s = new Stock("HR.A");
t = s;
r = new Stock("HR.B");
IO.println(s);
s.delimiter = '*';
IO.println(s); // HR.A*ACOMP CORP
IO.println(r); // HR.B BTECH INC.
```



(More) class usage

static vs. non-static

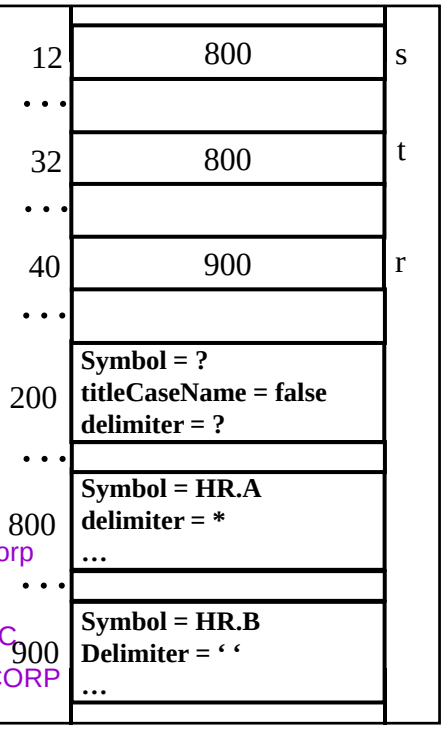
```
Stock s, t, r;
s = new Stock("HR.A");
t = s;
r = new Stock("HR.B");
IO.println(s);
s.delimiter = '*';
IO.println(s);
IO.println(r);
Stock.titleCaseName = true;
IO.println(s.getName()); //Acomp Corp
```



(More) class usage

static vs. non-static

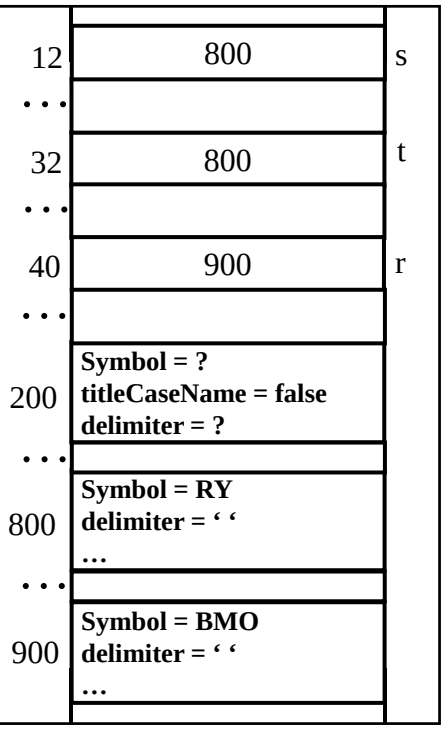
```
Stock s, t, r;
s = new Stock("HR.A");
t = s;
r = new Stock("HR.B");
IO.println(s);
s.delimiter = '*';
IO.println(s);
IO.println(r);
Stock.titleCaseName = true;
IO.println(s.getName()); // Acomp Corp
r.titleCaseName = false;
IO.println(r.getName()); //BTECH INC
IO.println(s.getName()); //ACOMP CORP
```



(More) class usage

Losing objects: 3 ways

```
Stock s, t, r;
s = new Stock("RY");
t = s;
r = new Stock("BMO");
```



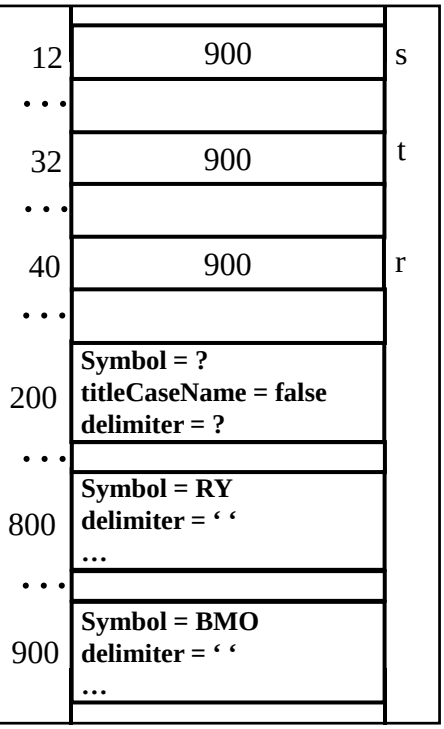
(More) class usage

Losing objects: 3 ways

```
Stock s, t, r;
s = new Stock("RY");
t = s;
r = new Stock("BMO");
```

1. Point elsewhere

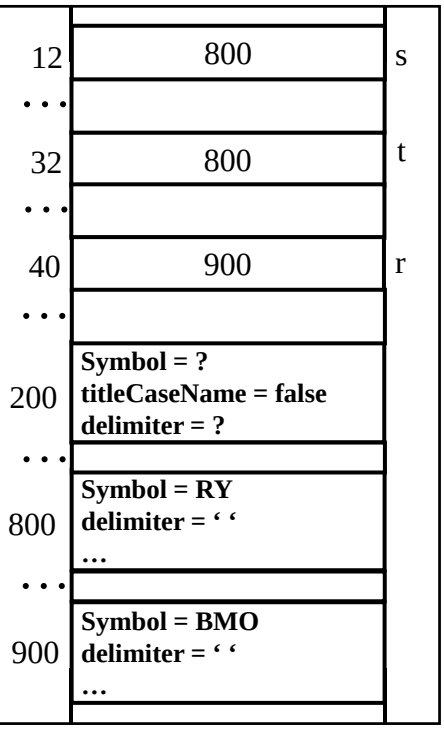
```
s = r;
t = r;
// now nothing points to 800
```



(More) class usage

Losing objects: 3 ways

```
Stock s, t, r;
s = new Stock("RY");
t = s;
r = new Stock("BMO");
```



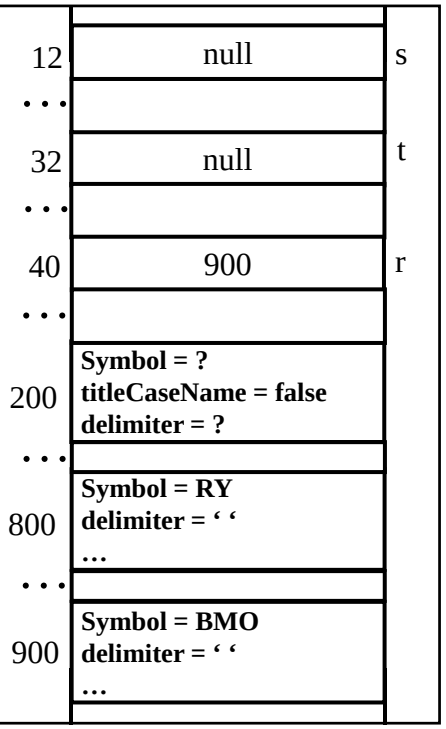
(More) class usage

Losing objects: 3 ways

```
Stock s, t, r;
s = new Stock("RY");
t = s;
r = new Stock("BMO");
```

2. Set to null

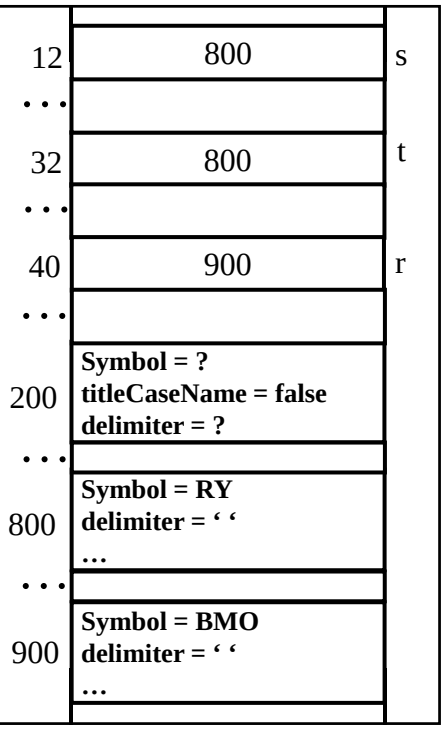
```
s = null;
t = null;
// now nothing points to 800
```



(More) class usage

Losing objects: 3 ways

```
{  Stock s, t, r;
   s = new Stock("RY");
   t = s;
   r = new Stock("BMO");
}
```

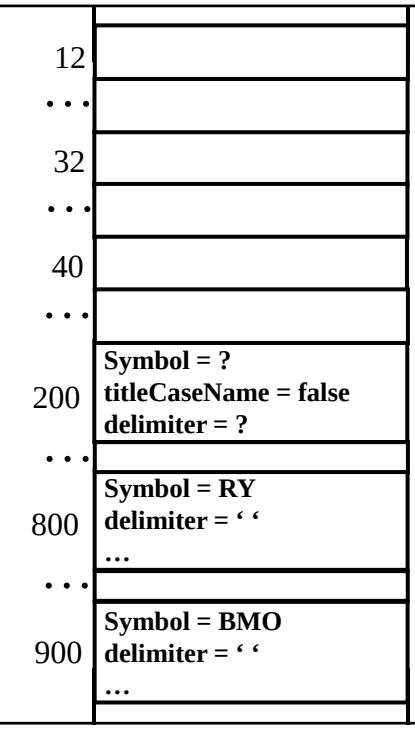


(More) class usage

Losing objects: 3 ways

```
{  Stock s, t, r;
   s = new Stock("RY");
   t = s;
   r = new Stock("BMO");
}
/* nothing points to 800 or 900
   here beyond the curly brackets
*/
```

3. Leave the scope.



(More) class usage

Parting comments

- Be sure that you understand class usage as illustrated by the previous examples.
- Create additional examples for yourself; try them out on your computer.
- Think in terms of memory diagrams to keep matters clear.

132

Outline

- Objects
- API and usage
- Patterns
- (More) usage
- **Abstraction: Objects, classes & methods**
- Software engineering example

Abstraction: Objects, classes & methods

Why Object Oriented Programming (OOP)?

- OOP lends itself to highly modular programs.
- OOP lends itself to high reuse of software.

Abstraction: Objects, classes & methods

Why make use of classes

- Data abstraction
- Modularity

135

Abstraction: Objects, classes & methods

Why make use of classes: Data abstraction

- We want to work with data pertaining to some entity.
  - For example, we might consider *stocks*…
  - …with *attributes* of *symbol* and *price*
- We find there are a related set of operations that allow us to manipulate the data
  - So for stocks, we might consider ways to *get/set symbols*…
  - …and ways to *get/refresh prices*
- It is useful to group together the data and the operations that manipulate it, e.g., into a class
  - Combining these operations with the data itself, we abstract to a class Stock.

136

Abstraction: Objects, classes & methods

Why make use of classes: Data abstraction

- To avoid confusion and inadvertent corruption of the data, we hide the details of
  - How the data is represented
  - How the operations are manipulated
- We provide a public interface to only a set of operations that allow the user access to critical pieces of information
  - For Stock, provide methods to alter symbols and prices.
  - But no direct access to the symbols and prices.
- Users of the class instantiate objects of the class and work through the (public) interface on this objects.
  - Create a Stock object myStock
  - Provide a symbol and price through methods

137

Abstraction: Objects, classes & methods

Why make use of classes: Modularity

- We want to group together a set of related operations in a module.
  - For example, the Math class.
  - The operations are instantiated as methods with public interfaces.
  - Again, details of operation are hidden from the user.
- In this case, users of the class
  - Will not instantiate objects of the class.
  - Instead, they work just with the methods through their public interfaces.

```
float x = Math.sin(Math.PI/2.0); // x has value 1.0
output.println("I hope you understand this.");
```

138

Abstraction: Objects, classes & methods

Why make use of classes

- Data abstraction
- Modularity
- In both cases, we say that the class encapsulates, i.e., hides, the details of its definition and implementation.

139

Abstraction: Objects, classes & methods

Methods provide procedural abstraction

- When we define a method, we take steps required to solve a subproblem and give them a name.
- Afterwards, the method can be called without knowing how it is implemented.

140

Abstraction: Objects, classes & methods

Methods provide procedural abstraction

- When we define a method, we take steps required to solve a subproblem and give them a name.
- Afterwards, the method can be called without knowing how it is implemented.

We can distinguish 2 types of methods

1. Instance (non-static) methods: Provide ways to operate on instances of a class, i.e., objects

```
int strLen = "How long am I?".length( );
```

2. Class (static) methods: Directly associated with the class; combined the methods in the class provide a functional module, e.g., the methods of the Math class.

```
double x = Math.abs(-2.0);
```

141

Abstraction: Objects, classes & methods

A class provides

- A declaration of the attributes (fields, variables, …)
  - Contain (static) data regarding the class as a whole (e.g., titleCaseName for Stock)
  - Contain (non-static) data regarding instances of the class (e.g., name and delimiter for a Stock object).
- Constructors
  - These are used to instantiate the object (in conjunction with the new operator).

```
Stock myStock = new Stock("BMO");
```

- Instance (non-static) methods
  - Those that operate on instances of the class (objects)

```
myStock.getPrice( );
```

142

Abstraction: Objects, classes & methods

A class provides (Cont.)

- Class (Static) methods
  - Those associated directly with the class

    x = Math.pow(2, 3); // 8.0

    or

    Stock.getExchangeCompIndexValue( ); // hypothetical
- Implementation
  - All the gory details.
  - But, *hidden away from the user* of the class.

143

Abstraction: Objects, classes & methods

A class provides (Cont.)

- Class (Static) methods
  - Those associated directly with the class

    x = Math.pow(2, 3); // 8.0

    or

    Stock.getExchangeCompIndexValue( ); // hypothetical
- Implementation
  - All the gory details.
  - But, *hidden away from the user* of the class.

Remark

- Static classes do *not* provide constructors, instance methods, instance attributes.

144

Abstraction: Objects, classes & methods

What we now understand

- Object oriented programming supports modular software development and high reuse.
- Classes provide data abstraction and modular design.
- Objects are instances of classes.
- Methods provide procedural abstraction.

145

Outline

- Objects
- API and usage
- Patterns
- (More) usage
- Abstraction: Objects, classes & methods
- **Software engineering example**

146

Software engineering example

Requirements: Problem definition

- We have been asked to write a program that generates lottery (lotto) numbers.

147

Software engineering example

Requirements: Analysis

- **Input:** No input is required.
- **Output:** Lottery (lotto) numbers, 6 random integers in the range 1-49.
- **Format:** i1 i2 i3 i4 i5 i6

148

Software engineering example

Design

149

Software engineering example

Design

- **Algorithm:**
  1. Generate a random number in [1,49].
  2. Print out the random number.
  3. Repeat steps 1 & 2 five more times.

150

Software engineering example

Design: What extant software can we use?

- Surely, someone has written a random number generator.

151

Software engineering example

Design: What extant software can we use?

- Surely, someone has written a random number generator.
- In java.util we find the class Random

public class Random
extends Object

An instance of this class is used to generate a stream of pseudorandom numbers. The class uses a 48-bit seed, which is modified using a linear congruential formula. (See Donald Knuth, The Art of Computer Programming, Volume 2, Section 3.2.1.)

152

Software engineering example

Design: What extant software can we use?

Constructor Summary

Random()
Creates a new random number generator.

Random(long seed)
Creates a new random number generator using a single long seed:

153

Software engineering example

Design: What extant software can we use?

Method summary

int	nextInt() Returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence.

154

Software engineering example

Design

- **Algorithm:**
  1. Generate a random number in [1,49].
  2. Print out the random number.
  3. Repeat steps 1 & 2 five more times.

155

Software engineering example

Design

- **Algorithm:**
  1. Generate a random number in [1,49].
     1.1 Generate an arbitrary random number
     1.2 Make the number non-negative.
     1.3 Scale the number to be in [0,48]
         (for integer n, n%49 in [0,48]).
     1.4 Shift the number to be in [1,49]
  2. Print the random number.
  3. Repeat steps 1 & 2 five more times.

156

Software engineering example

Design

- **Algorithm:**
  1. Generate a random number in [1,MAX].
     1.1 Generate an arbitrary random number
     1.2 Make the number non-negative.
     1.3 Scale the number to be in [0,MAX-1].
     1.4 Shift the number to be in [1,MAX]
  2. Print the random number.
  3. Repeat steps 1 & 2 five more times.
- **Constants:** 49 → MAX

157

Software engineering example

Design

- **Algorithm:**
  1. Generate a random number in [1,MAX].
     1.1 Generate an arbitrary random number
     1.2 Make the number non-negative.
     1.3 Scale the number to be in [0,MAX-1].
     1.4 Shift the number to be in [1,MAX]
  2. Print the random number.
  3. Repeat steps 1 & 2 five more times.
- **Constants:** 49 → MAX
- **Variables:** random number; generator.

158

Software engineering example

Example: Implementation:

- Declaration
- Input
- Computation
- Output

159

Software engineering example

Our 1020 program template

```
/*
Class to ???.

Author: ???                                    Date: ???
*/

import type.lang.*;
import java.util.Scanner;
import java.io.PrintStream;

// Definition of the ClassName class.
public class ClassName
{   public static void main(String[ ] args)
    {   Scanner input = new Scanner(System.in);
        PrintStream output = System.out;
        // App specific DICO.
    }
}
```

160

Software engineering example

Our 1020 program template

```
/*
Class to Class to generate lotto numbers: 6 random integers in
    [1,49].

Author: R. Wildes                                   Date: 05/09/13
*/

import type.lang.*;
import java.util.Scanner;
import java.io.PrintStream;
import java.util.Random; // import random number facilities

// Definition of the LottoNumbers class.
public class LottoNumbers
{    public static void main(String[ ] args)
     {    Scanner input = new Scanner(System.in);
          PrintStream output = System.out;
          // App specific DICO.
     }
}
```

161

Software engineering example

DICO: Declaration

```
// Declaration
```

162

Software engineering example

DICO: Declaration

// Declaration.

- During design we had isolated that we would want to manipulate data and/or objects having to do with
  - A random number (int)
  - A maximum value (int)
  - A generator of random values (Random)

163

Software engineering example

DICO: Declaration

// Declaration.

- During design we had isolated that we would want to manipulate data and/or objects having to do with
  - A random number (int)
  - A maximum value (int)
  - A generator of random values (Random)
- Some of these items should not change during the course of the program: Constants.

164

Software engineering example

DICO: Declaration

```
// Declaration.
final int MAX = 49;
```

165

Software engineering example

DICO: Declaration

// Declaration.

- During design we had isolated that we would want to manipulate data and/or objects having to do with
  - A random number (int)
  - A maximum value (int)
  - A generator of random values (Random)
- Some of these items should not change during the course of the program: Constants.
- Some of these items should have a primitive type and vary during the course of the program: primitive type variables.

166

Software engineering example

DICO: Declaration

```
// Declaration.
final int MAX = 49;
int rn;
```

167

Software engineering example

DICO: Declaration

// Declaration.

- During design we had isolated that we would want to manipulate data and/or objects having to do with
  - A random number (int)
  - A maximum value (int)
  - A generator of random values (Random)
- Some of these items should not change during the course of the program: Constants.
- Some of these items should be of primitive type and vary during the course of the program: primitive type variables.
- Some of these items should be our own copy of a predefined class: Objects type variables.

168

Software engineering example

DICO: Declaration

```
// Declaration.
final int MAX = 49;
int rn;
Random gen = new Random();
```

169

Software engineering example

DICO: Input

```
// Input.
```

170

Software engineering example

DICO: Input

// Input.

- **Input:** No input is required.

171

Software engineering example

DICO: Input

// Input: This app requires no input from user.

172

Software engineering example

DICO: Computation

// Computation.

173

Software engineering example

DICO: Computation

// Computation.

1. Generate a random number in [1,MAX].

174

Software engineering example

DICO: Computation

// Computation.
1. Generate a random number in [1,MAX].
    1.1 Generate an arbitrary random number

175

Software engineering example

DICO: Computation

// Computation.
rn = gen.nextInt();

176

Software engineering example

DICO: Computation

```
// Computation.
rn = gen.nextInt(); // get a random number
```

177

Software engineering example

DICO: Computation

```
// Computation.
```

1. Generate a random number in [1,MAX].
    1.1 Generate an arbitrary random number
    1.2 Make the number non-negative.

178

Software engineering example

DICO: Computation

```
// Computation
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
```

179

Software engineering example

DICO: Computation

// Computation

1. Generate a random number in [1,MAX].
    1.1 Generate an arbitrary random number
    1.2 Make the number non-negative.
    1.3 Scale the number to be in [0,MAX-1].

180

Software engineering example

DICO: Computation

```
// Computation.
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
rn = rn % MAX; // scale to be in [0,MAX-1]
```

181

Software engineering example

DICO: Computation

// Computation.

1. Generate a random number in [1,MAX].
    1.1 Generate an arbitrary random number
    1.2 Make the number non-negative.
    1.3 Scale the number to be in [0,MAX-1].
    1.4 Shift the number to be in [1,MAX]

182

Software engineering example

DICO: Computation

```
// Computation
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
rn = rn % MAX; // scale to be in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
```

183

Software engineering example

DICO: Computation

```
// Computation
```

1. Generate a random number in [1,MAX].
    1.1 Generate an arbitrary random number
    1.2 Make the number non-negative.
    1.3 Scale the number to be in [0,MAX-1].
    1.4 Shift the number to be in [1,MAX]
2. Print the random number.

184

Software engineering example

DICO: Computation and Output

// Computation and Output
1. Generate a random number in [1,MAX].
    1.1 Generate an arbitrary random number
    1.2 Make the number non-negative.
    1.3 Scale the number to be in [0,MAX-1].
    1.4 Shift the number to be in [1,MAX]
2. Print the random number.

Recall
• Output had the format: i1 i2 i3 i4 i5 i6

185

Software engineering example

DICO: Computation and Output

```
// Computation and Output
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
rn = rn % MAX; // scale to be in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 1
```

186

Software engineering example

DICO: Computation and Output

// Computation and Output

1. Generate a random number in [1,MAX].
    1.1 Generate an arbitrary random number
    1.2 Make the number non-negative.
    1.3 Scale the number to be in [0,MAX-1].
    1.4 Shift the number to be in [1,MAX]
2. Print the random number.
3. Repeat steps 1 & 2 five more times.

187

Software engineering example

Incremental edit/compile/run

- Prior to repeating the calculation...
- … let's pause to make sure that things are working as expected, so far.

188

Software engineering example

DICO: Computation and Output

```
// Computation and Output
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
rn = rn % MAX; // scale to be in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 1
```

Remark

- There is no need to enter debugging statements as the code segment ended with a useful output statement.

189

Software engineering example

Incremental edit/compile/run

- Prior to repeating the calculation...
- … let's pause to make sure that things are working as expected, so far.

%

190

Software engineering example

Incremental edit/compile/run

- Prior to repeating the calculation...
- … let's pause to make sure that things are working as expected, so far.

```
% javac LottoNumbers.java
```

191

Software engineering example

Incremental edit/compile/run

- Prior to repeating the calculation...
- … let's pause to make sure that things are working as expected, so far.

```
% javac LottoNumbers.java
%
```

192

Software engineering example

Incremental edit/compile/run

- Prior to repeating the calculation...
- … let's pause to make sure that things are working as expected, so far.

```
% javac LottoNumbers.java
% java LottoNumbers
```

193

Software engineering example

Incremental edit/compile/run

- Prior to repeating the calculation...
- … let's pause to make sure that things are working as expected, so far.

```
% javac LottoNumbers.java
% java LottoNumbers
39
```

194

Software engineering example

DICO: Computation and Output

// Computation and Output
1. Generate a random number in [1,MAX].
    1.1 Generate an arbitrary random number
    1.2 Make the number non-negative.
    1.3 Scale the number to be in [0,MAX-1].
    1.4 Shift the number to be in [1,MAX]
2. Print the random number.
3. Repeat steps 1 & 2 five more times.

195

Software engineering example

DICO: Computation and Output

// Computation and Output
1. Generate a random number in [1,MAX].
    1.1 Generate an arbitrary random number
    1.2 Make the number non-negative.
    1.3 Scale the number to be in [0,MAX-1].
    1.4 Shift the number to be in [1,MAX]
2. Print the random number.
3. Repeat steps 1 & 2 five more times.

Remark
- There is a better way.

196

Software engineering example

Design: What extant software can we use?

Method summary

int	nextInt() Returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence.
int	nextInt(int n) Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

197

Software engineering example

DICO: Computation and Output

// Computation and Output

1. Generate a random number in [1,MAX].
    1.1 Generate an arbitrary random number
    1.2 Make the number non-negative.
    1.3 Scale the number to be in [0,MAX-1].
    1.4 Shift the number to be in [1,MAX]
2. Print the random number.
3. Repeat steps 1 & 2 five more times.

Remark

- There is a better way.

198

Software engineering example

DICO: Computation and Output

// Computation and Output
1. Generate a random number in [1,MAX].
    1.1 Generate an random number in [0,MAX-1].
    1.2 Shift the number to be in [1,MAX].
2. Print the random number.
3. Repeat steps 1 & 2 five more times.

199

Software engineering example

DICO: Computation and Output

```
// Computation and Output
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
rn = rn % MAX; // scale to be in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 1
```

200

Software engineering example

DICO: Computation and Output

```
// Computation and Output
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
rn = rn % MAX; // scale to be in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 1
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
```

201

Software engineering example

DICO: Computation and Output

```
// Computation and Output
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
rn = rn % MAX; // scale to be in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 1
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
```

202

Software engineering example

DICO: Computation and Output

```
// Computation and Output
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
rn = rn % MAX; // scale to be in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 1
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 2
```


Software engineering example

DICO: Computation and Output

```
// Computation and Output
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
rn = rn % MAX; // scale to be in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 1
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 2
```

Remark

- Now is a good time for incremental edit/compile/run.

Software engineering example

DICO: Computation and Output

// Computation and Output
1. Generate a random number in [1,MAX].
    1.1 Generate an random number in [0,MAX-1].
    1.2 Shift the number to be in [1,MAX].
2. Print the random number.
3. Repeat steps 1 & 2 five more times.

205

Software engineering example

DICO: Computation and Output

```
// Computation and Output
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
rn = rn % MAX; // scale to be in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 1
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 2
```

206

Software engineering example

DICO: Computation and Output

```
// Computation and Output
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
rn = rn % MAX; // scale to be in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 1
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 2
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 3
```

207

Software engineering example

DICO: Computation and Output

```
// Computation and Output
rn = gen.nextInt(); // get a random number
rn = Math.abs(rn); // make non-negative
rn = rn % MAX; // scale to be in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 1
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 2
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 3
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 4
```

208

Software engineering example

DICO: Computation and Output

```
// Computation and Output
.
.
.
```

209

Software engineering example

DICO: Computation and Output

```
// Computation and Output
.
.
.
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 5
```

210

Software engineering example

DICO: Computation and Output

```
// Computation and Output
.
.
.
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.print(rn + " "); // that's 5
rn = gen.nextInt(MAX); // get random in [0,MAX-1]
rn = rn + 1; // shift to be in [1,MAX]
output.println(rn); // that's 6
```

Remark

- Notice that the last printing statement is slightly different than the previous ones.

211

Software engineering example

Completing the implementation cycle

- We now save our code to a file LottoNumbers.java…
- …and continue with the edit/compile/run cycle until
- …we have nominally working LottoNumbers.class
- Since we have been working incrementally through the edit/compile/run cycle, final compilation should go relatively smoothly.

212

Software engineering example

Test

%

213

Software engineering example

Test

% java LottoNumbers

214

Software engineering example

Test

```
% java LottoNumbers
11 42 18 24 3 7
%
```

215

Software engineering example

Test

```
% java LottoNumbers
11 42 18 24 3 7
% java LottoNumbers
```

216

Software engineering example

Test

```
% java LottoNumbers
11 42 18 24 3 7
% java LottoNumbers
31 36 4 46 43 39
%
```

217

Software engineering example

Test

```
% java LottoNumbers
11 42 18 24 3 7
% java LottoNumbers
31 36 4 46 43 39
% java LottoNumbers
```

218

Software engineering example

Test

```
% java LottoNumbers
11 42 18 24 3 7
% java LottoNumbers
31 36 4 46 43 39
% java LottoNumbers
25 33 26 38 41 37
%
```

219

Software engineering example

Test

```
% java LottoNumbers
11 42 18 24 3 7
% java LottoNumbers
31 36 4 46 43 39
% java LottoNumbers
25 33 26 38 41 37
%
```

Remark

- In practice, would submit program to a more extensive battery of tests.

220

Software engineering example

Deployment

- In real life you now ship/install your product.
- Here, as usual, I've placed the source code on our section website.

221

Summary

- **Objects**
- **API and usage**
- **Patterns**
- **(More) usage**
- **Abstraction: Objects, classes & methods**
- **Software engineering example**

222