

Topic 3

Encapsulation - Implementing Classes

“And so, from Europe, we get things such as ... object-oriented analysis and design (a clever way of breaking up software programming instructions and data into small, reusable objects, based on certain abstraction principles and design hierarchies.)”

*-Michael A. Cusumano,
The Business Of Software*



Object Oriented Programming

- ▶ Creating large programs that work turns out to be very difficult
 - DIA Automated baggage handling system
 - Ariane 5 Flight 501
 - More
- ▶ Object oriented programming is one way of *managing the complexity* of programming and software projects
- ▶ Break up big problems into smaller, more manageable problems

Object Oriented Programming

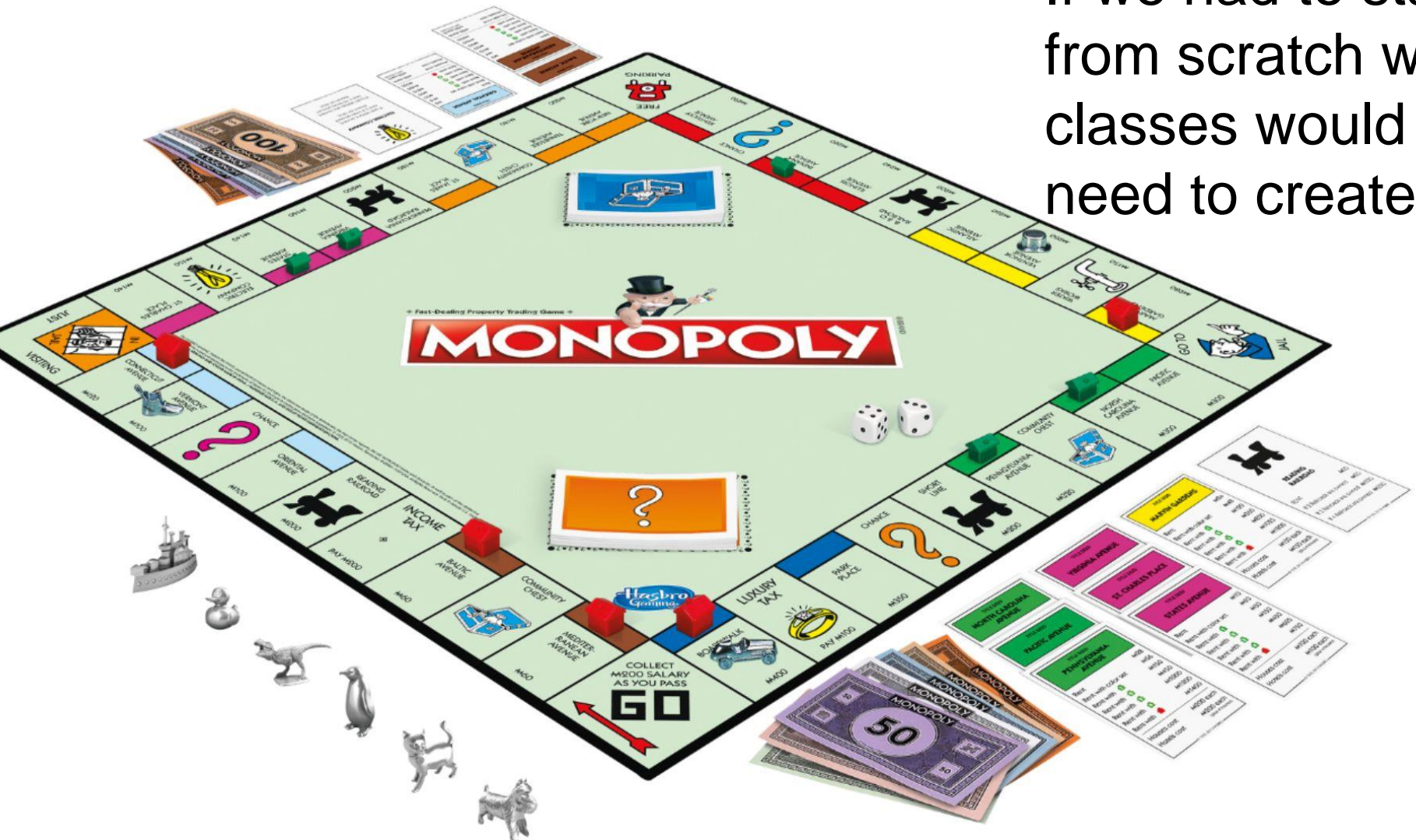
- ▶ "Object-oriented programming is a method of programming based on a hierarchy of classes, and well-defined and cooperating objects. "
- ▶ What is a class?
- ▶ "A class is a structure that defines the data and the methods to work on that data. When you write programs in the Java language, all program data is wrapped in a class, whether it is a class you write or a class you use from the Java platform API libraries."
 - a new data type

Object Oriented Programming

- ▶ In other words break the problem up based on the things / data types that are part of the problem
- ▶ Not the only way
- ▶ One of many different kinds of strategies or *paradigms* for software development
 - functional, procedural, event driven, data flow, formal methods, agile or extreme, ...
- ▶ In 314 we will do a lot of *object based* programming

Example - Monopoly

If we had to start from scratch what classes would we need to create?



Encapsulation

- ▶ One of the features of object oriented languages
- ▶ Allows programmers to define **new data types**
- ▶ Hide the data of an object (variable)
- ▶ Group operations and data together into a new data type
- ▶ Usually easier to ***use*** something than understand ***exactly how it works***
 - microwave, car, computer, software, mp3 player

Data Structures

- ▶ A data structure is a variable that stores other variables. (overly simplified definition)
 - aka Collection, Container
- ▶ May be ordered or unordered (from client's perspective)
 - Order a first element, second element,...
 - Lists are ordered, sets are typically unordered
- ▶ May allow duplicate values or not
 - Lists allow duplicates, sets typically do not

The IntList Class

- ▶ We will develop a class that models a list of ints
 - initially a pale imitation of the Java ArrayList class
 - ▶ Improvement on an array of ints
 - resize automatically
 - insert easily
 - remove easily
 - ▶ A list - our first *data structure*
 - a variable that stores other variables
 - ▶ Lists maintain elements in a definite order and duplicates are allowed
- ```
0 1 2 3 4 <- indices / positions
[5, 12, 5, 17, -5] <- elements
```



# Clicker 1

Our `IntList` class has an array of ints instance variable (`int[] container`). What should the length of this internal array be?

- A. less than the size of the list
- B. equal to the size of the list
- C. greater than or equal to the size of the list
- D. some fixed amount that never changes
- E. 0

# Array length less than the number of elements in the list?!?

- ▶ What if most elements are all the same value? Only store the elements (and their position) not equal to the default? Sparse List



## Clicker 2

When adding a new element to a list, where should the new element be added by default?

- A. The beginning
- B. The end
- C. The middle
- D. A random location
- E. Don't bother to actually add

# IntList Design

- ▶ Create a new, empty IntList

`new IntList -> []`

- ▶ The above is not code. It is a notation that shows what the results of operations. `[]` is an empty list.
- ▶ add to a list.

`[] .add(1) -> [1]`

`[1] .add(5) -> [1, 5]`

`[1, 5] .add(4) -> [1, 5, 4]`

- ▶ elements in a list have a definite order and a position.
  - zero based position or 1 based positioning?

```
IntList aList = new IntList();
aList.add(42);
aList.add(12);
aList.add(37);
```

Abstract view of  
list of integers

|      |     |     |
|------|-----|-----|
| 0    | 1   | 2   |
| [42, | 12, | 37] |

The wall of  
abstraction.

# Instance Variables

- ▶ Internal data
  - also called instance variables because every instance (object) of this class has its own copy of these
  - something to store the elements of the list
  - size of internal storage container?
  - if not what else is needed
- ▶ Must be clear on the difference between the internal data of an IntList object and the IntList that is being represented
- ▶ Why make internal data private?

# Constructors

- ▶ For initialization of objects
- ▶ IntList constructors
  - default
  - initial capacity?
- ▶ redirecting to another constructor  
`this(10);`
- ▶ class constants
  - what `static` means

# Default add method

- ▶ where to add?
- ▶ what if not enough space?

`[] .add(3) -> [3]`

`[3] .add(5) -> [3, 5]`

`[3, 5] .add(3) -> [3, 5, 3]`

- ▶ Testing, testing, testing!
  - a `toString` method would be useful



# The IntList Class

- ▶ instance variables
- ▶ constructors
  - default
  - initial capacity
    - preconditions, exceptions, postconditions, assert
  - meaning of static
- ▶ add method
- ▶ get method
- ▶ size method

# toString method

- ▶ return a Java String of list
- ▶ empty list -> []
- ▶ one element -> [12]
- ▶ multiple elements -> [12, 0, 5, 4]

## Clicker 3 - Timing Experiment

- ▶ Add N elements to an initially empty IntList then call toString. Time both events. How does the time to add compare to the time to complete toString?

```
IntList list = new IntList();
for (int i = 0; i < N; i++)
 list.add(i); // resize, cap * 2
String s = list.toString();
```

- A. time to add << time for toString()
- B. time to add < time for toString()
- C. time to add ≈ time for toString()
- D. time to add > time for toString()
- E. time to add >> time for toString()

# The IntList Class

- ▶ testing!!!
- ▶ toString
  - “beware the performance of String concatenation” – Joshua Bloch
- ▶ insert method (`int pos, int value`)
- ▶ remove method (`int pos`)
- ▶ insertAll method  
(`int pos, IntList other`)
  - queens and kings of all the IntLists!!!

# Clicker Question 4

What is output by the following code?

```
IntList list
list = new IntList(25);
System.out.println(list.size());
```

A. 25

B. 0

C. -1

D. unknown

E. No output due to runtime error.

# get and size methods

## ▶ get

- access element from list
- preconditions?

`[3, 5, 2].get(0)` returns 3

`[3, 5, 2].get(1)` returns 5

## ▶ size


- number of elements in the list
- Do not confuse with the capacity of the internal storage container
- The array is not the list!

`[4, 5, 2].size()` returns 3

# insert method

- ▶ add at someplace besides the end

`[3, 5].insert(1, 4) -> [3, 4, 5]`



where      what

`[3, 4, 5].insert(0, 4) -> [4, 3, 4, 5]`

- ▶ preconditions?
- ▶ overload add?
- ▶ chance for internal loose coupling

# Clicker 5

What is output by the following code?

```
IntList list = new IntList();
list.add(3);
list.insert(0, 4); // position, value
list.insert(1, 1);
list.add(5);
list.insert(2, 9);
System.out.println(list);
```

- A. [4, 1, 3, 9, 5]
- B. [3, 4, 1, 5, 9]
- C. [4, 1, 9, 3, 5]
- D. [3, 1, 4, 9, 5]
- E. Something else



# remove method

- ▶ remove an element from the list based on location

`[3, 4, 5].remove(0) -> [4, 5]`

`[3, 5, 6, 1, 2].remove(2) ->`  
`[3, 5, 1, 2]`

- ▶ preconditions?
- ▶ return value?
  - accessor methods, mutator methods, and mutator methods that return a value

# Clicker Question 6

What is output by the following code?

```
IntList list = new IntList();
list.add(12);
list.add(15);
list.add(12);
list.add(17);
list.remove(1);
System.out.println(list);
```

- A. [15, 17]
- B. [12, 17]
- C. [12, 0, 12, 17]
- D. [12, 12, 17]
- E. [15, 12, 17]

# insertAll method

- ▶ add all elements of one list to another starting at a specified location

```
[5, 3, 7].insertAll(2, [2, 3]) ->
[5, 3, 2, 3, 7]
```

The parameter `[2, 3]` would be unchanged.

- ▶ Working with other objects of the same type
  - `this`?
  - where is private private?
  - loose coupling vs. performance
  - queens and kings of all the `IntLists`!!!

# Clicker 7 - InsertAll First Version


- ▶ What is the order of the first version of InsertAll? Assume both lists have  $N$  elements and that the insert position is halfway through the calling list.

- A.  $O(1)$
- B.  $O(\log N)$
- C.  $O(N^{0.5})$
- D.  $O(N)$
- E.  $O(N^2)$

# Class Design and Implementation – Another Example

This example will not be covered  
in class.

# The Die Class

- ▶ Consider a class used to model a die
  - ▶ What is the interface? What actions should a die be able to perform?
- 
- ▶ The methods or behaviors can be broken up into constructors, mutators, accessors

# The Die Class Interface

- ▶ Constructors (used in creation of objects)
  - default, single int parameter to specify the number of sides, int and boolean to determine if should roll
- ▶ Mutators (change state of objects)
  - roll
- ▶ Accessors (do not change state of objects)
  - getResult, getNumSides, toString
- ▶ Public constants
  - DEFAULT\_SIDES

# Visibility Modifiers

- ▶ All parts of a *class* have visibility modifiers
  - Java keywords
  - **public**, protected, **private**, (no modifier means package access)
  - do not use these modifiers on local variables (syntax error)
- ▶ **public** means that constructor, method, or field may be accessed outside of the class.
  - part of the interface
  - constructors and methods are generally public
- ▶ **private** means that part of the class is hidden and inaccessible by code outside of the class
  - part of the implementation
  - data fields are generally private



# The Die Class Implementation

- ▶ Implementation is made up of constructor code, method code, and private data members of the class.
- ▶ scope of data members / instance variables
  - *private data members may be used in any of the constructors or methods of a class*
- ▶ Implementation is hidden from users of a class and can be changed without changing the interface or affecting clients (other classes that use this class)
  - Example: Previous version of Die class, DieVersion1.java
- ▶ Once Die class completed can be used in anything requiring a Die or situation requiring random numbers between 1 and N
  - DieTester class. What does it do?

# DieTester method

```
public static void main(String[] args) {
 final int NUM_ROLLS = 50;
 final int TEN_SIDED = 10;
 Die d1 = new Die();
 Die d2 = new Die();
 Die d3 = new Die(TEN_SIDED);
 final int MAX_ROLL = d1.getNumSides() +
 d2.getNumSides() + d3.getNumSides();

 for(int i = 0; i < NUM_ROLLS; i++)
 {
 d1.roll();
 d2.roll();
 System.out.println("d1: " + d1.getResult()
 + " d2: " + d2.getResult() + " Total: "
 + (d1.getResult() + d2.getResult()));
 }
}
```

# DieTester continued

```
int total = 0;
int numRolls = 0;
do
{
 d1.roll();
 d2.roll();
 d3.roll();
 total = d1.getResult() + d2.getResult()
 + d3.getResult();
 numRolls++;
}
while(total != MAX_ROLL);

System.out.println("\n\nNumber of rolls to get "
 + MAX_ROLL + " was " + numRolls);
```

# Correctness Sidetrack

- ▶ When creating the public interface of a class give careful thought and consideration to the *contract* you are creating between yourself and users (other programmers) of your class
- ▶ Use *preconditions* to state what you assume to be true before a method is called
  - caller of the method is responsible for making sure these are true
- ▶ Use *postconditions* to state what you guarantee to be true after the method is done if the preconditions are met
  - implementer of the method is responsible for making sure these are true

# Precondition and Postcondition Example

```
/* pre: numSides > 1
 post: getResult() = 1, getNumSides() = sides
*/
public Die(int numSides)
{
 assert (numSides > 1) : "Violation of precondition: Die(int)";
 iMyNumSides = numSides;
 iMyResult = 1;
 assert getResult() == 1 && getNumSides() == numSides;
}
```

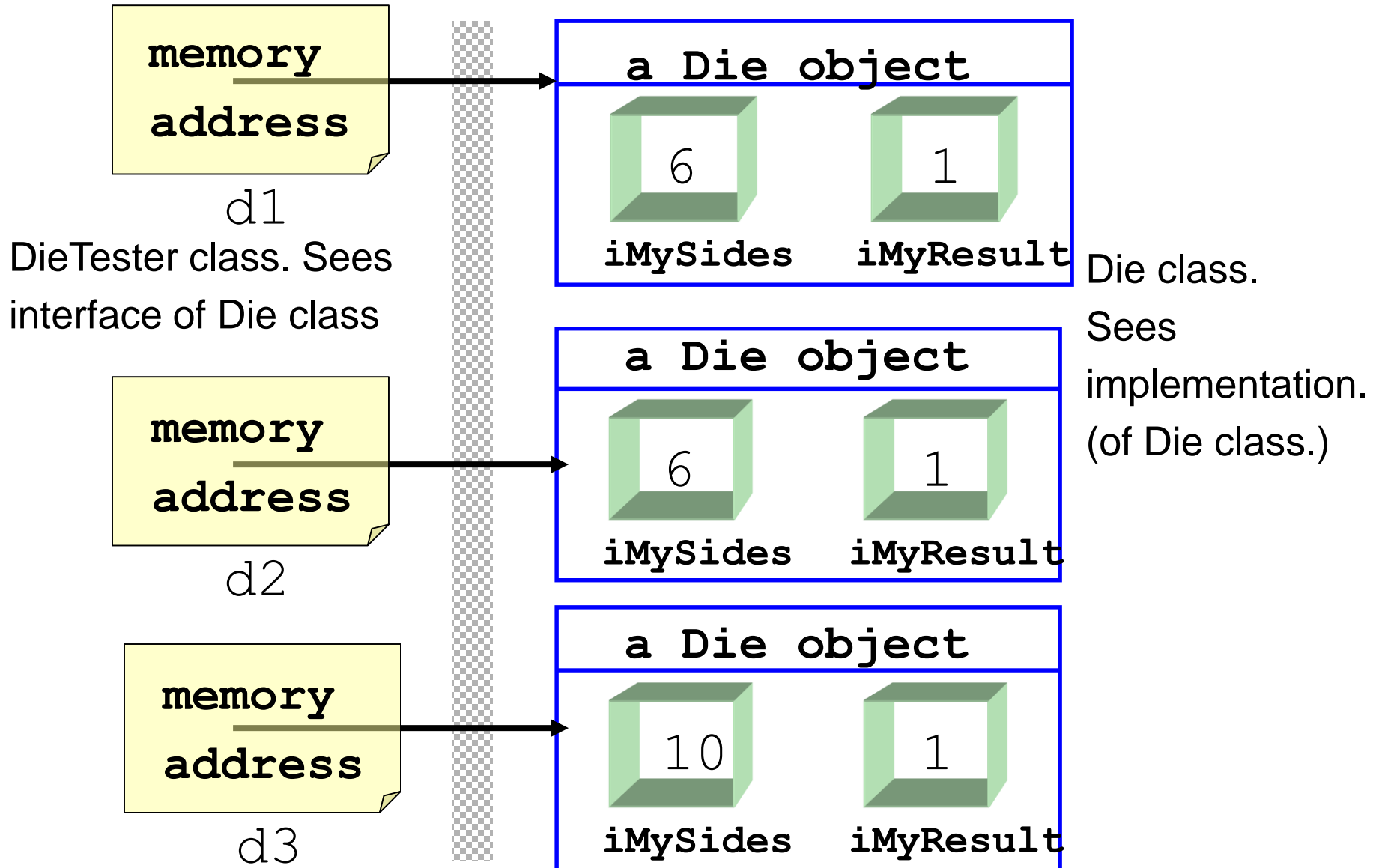
# Object Behavior - Instantiation

- ▶ Consider the DieTester class

```
Die d1 = new Die();
Die d2 = new Die();
Die d3 = new Die(10);
```

- ▶ When the new operator is invoked control is transferred to the Die class and the specified constructor is executed, based on parameter matching
- ▶ Space(memory) is set aside for the new object's fields
- ▶ The memory address of the new object is passed back and stored in the object variable (pointer)
- ▶ After creating the object, methods may be called on it.

# Creating Dice Objects



# Objects

- ▶ Every Die object created has its own instance of the variables declared in the class blueprint

```
private int iMySides;
private int iMyResult;
```

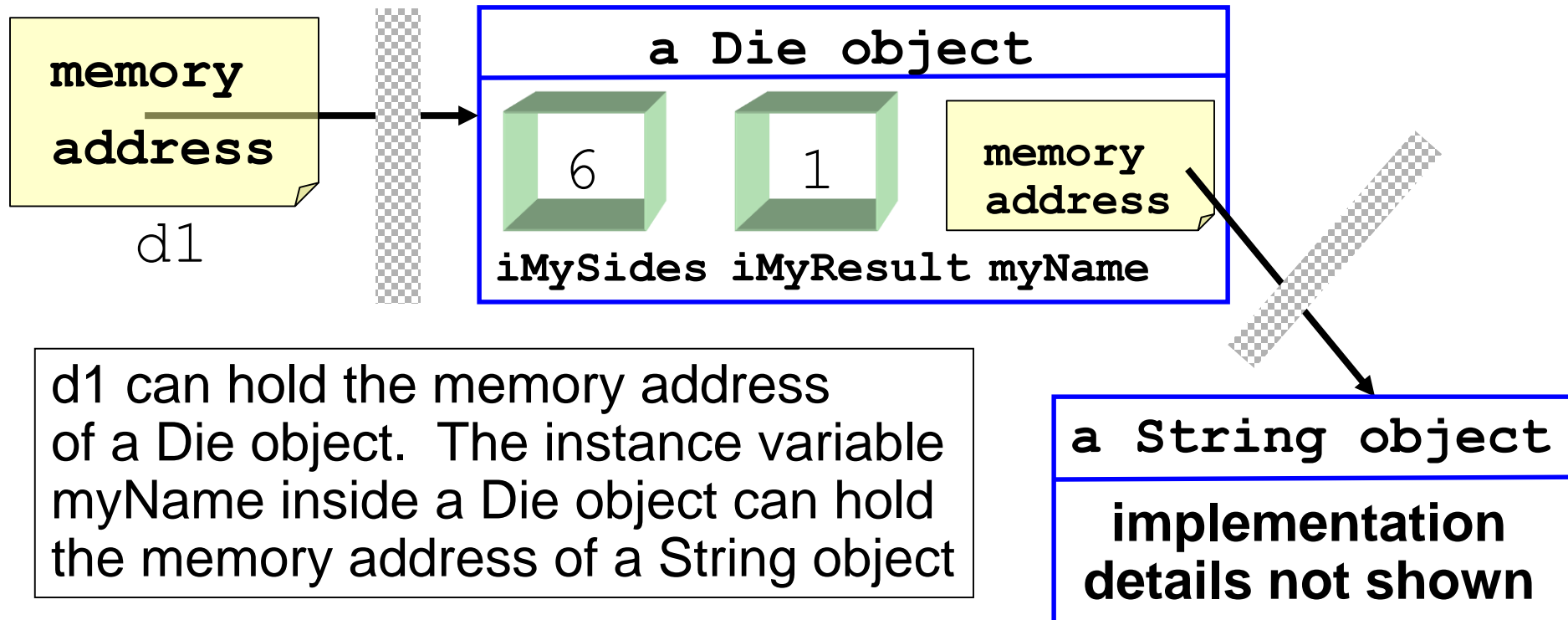
- ▶ thus the term *instance variable*
- ▶ the instance vars are part of the hidden implementation and may be of *any* data type
  - unless they are public, which is almost always a bad idea if you follow the tenets of information hiding and encapsulation



# Complex Objects

- ▶ What if one of the instance variables is itself an object?
- ▶ add to the Die class

```
private String myName;
```



# The Implicit Parameter

- ▶ Consider this code from the Die class

```
public void roll()
{ iMyResult =
 ourRandomNumGen.nextInt(iMySides) + 1;
}
```

- ▶ Taken in isolation this code is rather confusing.
- ▶ what is this iMyResult thing?
  - It's not a parameter or local variable
  - why does it exist?
  - *it belongs to the Die object that called this method*
  - if there are numerous Die objects in existence
  - Which one is used depends on which object called the method.

# The *this* Keyword

- ▶ When a method is called it may be necessary for the calling object to be able to refer to itself
  - most likely so it can pass itself somewhere as a parameter
- ▶ when an object calls a method an implicit reference is assigned to the calling object
- ▶ the name of this implicit reference is `this`
- ▶ `this` is a reference to the current calling object and may be used as an object variable (may not declare it)

# *this* Visually

```
// in some class other than Die
Die d3 = new Die();
d3.roll();
```

```
// in the Die class
public void roll()
{ iMyResult =
 ourRandomNumGen.nextInt(iMySides) + 1;
 /* OR
 this.iMyResult...
 */
}
```

memory  
address

this

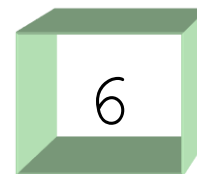


memory  
address

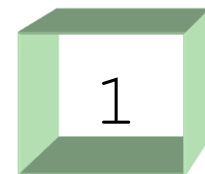
d3



a Die object



iMySides



iMyResult

# An equals method

- ▶ working with objects of the same type in a class can be confusing
- ▶ write an equals method for the Die class.  
assume every Die has a myName instance variable as well as iMyNumber and iMySides

# A Possible Equals Method

```
public boolean equals(Object otherObject)
{
 Die other = (Die)otherObject;
 return iMySides == other.iMySides
 && iMyResult == other.iMyResult
 && myName.equals(other.myName);
}
```

- ▶ Declared Type of Parameter is Object not Die
- ▶ override (replace) the equals method instead of overload (present an alternate version)
  - easier to create generic code
- ▶ we will see the equals method is *inherited* from the Object class
- ▶ access to another object's private instance variables?

# Another equals Methods

```
public boolean equals(Object otherObject)
{
 // dangerous! Not checking for null or type.
 Die other = (Die)otherObject;
 return this.iMySides == other.iMySides
 && this.iMyNumber == other.iMyNumber
 && this.myName.equals(other.myName);
}
```

---

Using the `this` keyword / reference to access the implicit parameters instance variables is unnecessary.

If a method within the same class is called within a method, the original calling object is still the calling object

# A "Perfect" Equals Method

## ► From Cay Horstmann's *Core Java*

```
public boolean equals(Object otherObject)
{
 // check if objects identical
 if(this == otherObject)
 return true;
 // must return false if explicit parameter null
 if(otherObject == null)
 return false;
 // if objects not of same type they cannot be equal
 if(getClass() != otherObject.getClass())
 return false;
 // we know otherObject is a non null Die
 Die other = (Die)otherObject;
 return iMySides == other.iMySides
 && iMyNumber == other.iMyNumber
 && myName.equals(other.myName);
}
```



# the instanceof Operator

- ▶ `instanceof` is a Java keyword.

- ▶ part of a boolean statement

```
public boolean equals(Object otherObj)
{
 if otherObj instanceof Die
 {
 //now go and cast
 // rest of equals method
 }
}
```

- ▶ Should not use `instanceof` in `equals` methods.
- ▶ `instanceof` has its uses but not in `equals` because of the contract of the `equals` method

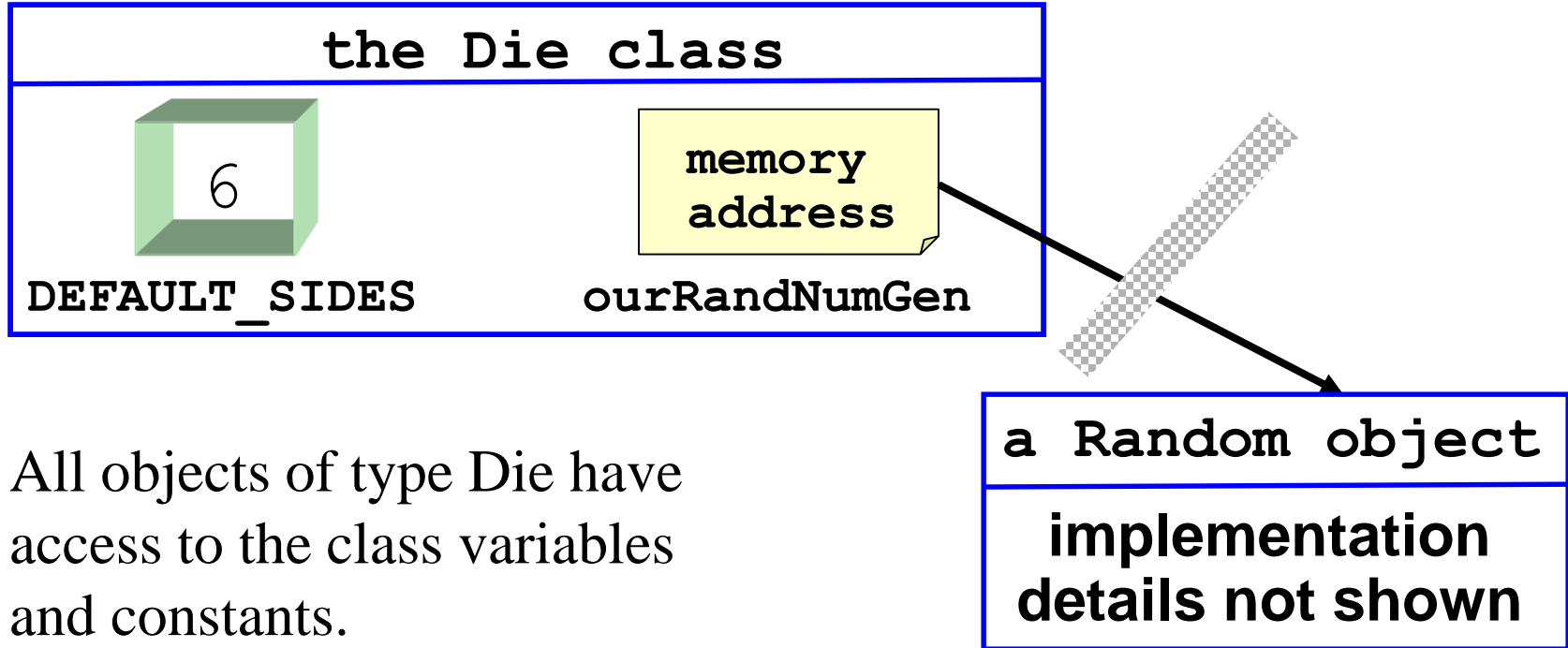
# Class Variables and Class Methods

- ▶ Sometimes every object of a class does not need its own copy of a variable or constant
- ▶ The keyword `static` is used to specify class variables, constants, and methods

```
private static Random ourRandNumGen
 = new Random();
public static final int DEFAULT_SIDES = 6;
```

- ▶ The most prevalent use of `static` is for class constants.
  - if the value can't be changed why should every object have a copy of this non changing value

# Class Variables and Constants



All objects of type Die have access to the class variables and constants.

A public class variable or constant may be referred to via the class name.

# Syntax for Accessing Class Variables

```
public class UseDieStatic
{ public static void main(String[] args)
 { System.out.println("Die.DEFAULT_SIDES "
 + Die.DEFAULT_SIDES);
 // Any attempt to access Die.ourRandNumGen
 // would generate a syntax error

 Die d1 = new Die(10);

 System.out.println("Die.DEFAULT_SIDES "
 + Die.DEFAULT_SIDES);
 System.out.println("d1.DEFAULT_SIDES "
 + d1.DEFAULT_SIDES);

 // regardless of the number of Die objects in
 // existence, there is only one copy of DEFAULT_SIDES
 // in the Die class

 } // end of main method
} // end of UseDieStatic class
```

# Static Methods

- ▶ `static` has a somewhat different meaning when used in a method declaration
- ▶ static methods may not manipulate any instance variables
- ▶ in non static methods, some object invokes the method  
`d3.roll()` ;
- ▶ the object that makes the method call is an implicit parameter to the method

# Static Methods Continued

- ▶ Since there is no implicit object parameter sent to the static method it does not have access to a copy of any objects instance variables
  - unless of course that object is sent as an explicit parameter
- ▶ Static methods are normally utility methods or used to manipulate static variables ( class variables )
- ▶ The Math and System classes are nothing but static methods

# static and this

- ▶ Why does this work (added to Die class)

```
public class Die
{
 public void outputSelf()
 { System.out.println(this);
 }
}
```

- ▶ but this doesn't?

```
public class StaticThis
{
 public static void main(String[] args)
 { System.out.println(this);
 }
}
```