Objects and memory

The stack
The heap
Variable life and death
null
Clones
Serializing objects
Memory: A programmer’s prospective

- Recall: Memory can be viewed as a sequence of numbered ‘storage boxes’
  - Essentially memory is an array
- The compiler decides (on our behalf) where variables will live in memory (their address)

```java
public static void main (String[] args) {
    int x = 0;
    char c = 'a';
    double dx = 140000000000;
    ...
} // end method main
```

Symbol Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Length</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>int</td>
<td>1</td>
<td>x0001</td>
</tr>
<tr>
<td>c</td>
<td>char</td>
<td>1</td>
<td>x0002</td>
</tr>
<tr>
<td>dx</td>
<td>double</td>
<td>2</td>
<td>x0003</td>
</tr>
<tr>
<td>xFFFD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xFFE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xFFFF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory: A programmer’s prospective

- Areas of memory are allocated for the different types of things that need to be stored
  - Operating System routines
  - The executable code
  - Static/global variables
  - Dynamically created objects
    - Instance variables
    - Memory needs grow/shrink during execution of the program.
    - The compiler has no idea how much space you might need (may depend on input, etc)
  - Local variables (a.k.a. stack variables)
    - Memory requirements change as variables local to the method come into and out of scope

- The amount of available memory is system dependent
Types of variables

public class Main {
    public static int number;
    public static void main (String[] args) {
        System.out.println(number);
    } // end method main
} // end class Main

Static/global variable: number

public class Point {
    int x;
    int y;
} // end class Point

Instance variables: x, y

public int foo (int bar){
    int temp = bar + 1;
    return temp;
} // end class Point

Local/stack variables: bar, temp
public class Point {
    int x;
    int y;
    Point (int xValue, int yValue) {
        x = xValue;
        y = yValue;
    } // end constructor

    Point makePoint (int x, int y) {
        Point p1 = new Point(x,y);
        return p1;
    } // end method Point

    static void main (String[] args) {
        Point p1 = new Point(1,2);
        Point p2 = makePoint(3,4);
        } end method main
} // end class Point
public class Point {
    int x;
    int y;
    Point (int xValue, int yValue) {
        x = xValue;
        y = yValue;
    } // end constructor

    Point makePoint (int x, int y) {
        Point p1 = new Point(x,y);
        return p1;
    } // end method Point

    static void main (String[] args) {
        Point p1 = new Point(1,2);
        Point p2 = makePoint(3,4);
    } end method main
} // end class Point

After “new” but before constructor is executed
public class Point {
    int x;
    int y;
    Point (int xValue, int yValue) {
        x = xValue;
        y = yValue;
    } // end constructor

    Point makePoint (int x, int y) {
        Point p1 = new Point(x,y);
        return p1;
    } // end method Point

    static void main (String[] args) {
        Point p1 = new Point(1,2);
        Point p2 = makePoint(3,4);
    } end method main
} // end class Point
public class Point {
    int x;
    int y;
    Point (int xValue, int yValue) {
        x = xValue;
        y = yValue;
    } // end constructor

    Point makePoint (int x, int y) {
        Point p1 = new Point(x,y);
        return p1;
    } // end method Point

    static void main (String[] args) {
        Point p1 = new Point(1,2);
        Point p2 = makePoint(3,4);
    } // end method main
} // end class Point

constructor is about to finish
public class Point {
    int x;
    int y;
    Point (int xValue, int yValue) {
        x = xValue;
        y = yValue;
    } // end constructor

    Point makePoint (int x, int y) {
        Point p1 = new Point(x,y);
        return p1;
    } // end method Point

    static void main (String[] args) {
        Point p1 = new Point(1,2);
        Point p2 = makePoint(3,4);
    } end method main
} // end class Point

Execution of the first line of main complete
public class Point {
    int x;
    int y;
    Point (int xValue, int yValue) {
        x = xValue;
        y = yValue;
    } // end constructor

    Point makePoint (int x, int y) {
        Point p1 = new Point(x,y);
        return p1;
    } // end method Point

    static void main (String[] args) {
        Point p1 = new Point(1,2);
        Point p2 = makePoint(3,4);
    } end method main
} // end class Point

Preparing to execute makePoint
public class Point {
    int x;
    int y;
    Point (int xValue, int yValue) {
        x = xValue;
        y = yValue;
    } // end constructor

    Point makePoint (int x, int y) {
        Point p1 = new Point(x,y);
        return p1;
    } // end method Point

    static void main (String[] args) {
        Point p1 = new Point(1,2);
        Point p2 = makePoint(3,4);
    } end method main
} // end class Point
public class Point {
    int x;
    int y;
    Point (int xValue, int yValue) {
        x = xValue;
        y = yValue;
    } // end constructor

    Point makePoint (int x, int y) {
        Point p1 = new Point(x,y);
        return p1;
    } // end method Point

    static void main (String[] args) {
        Point p1 = new Point(1, 2);
        Point p2 = makePoint(3, 4);
    } end method main
} // end class Point

makePoint near end of constructor call
public class Point {
    int x;
    int y;
    Point (int xValue, int yValue) {
        x = xValue;
        y = yValue;
    } // end constructor

    Point makePoint (int x, int y) {
        Point p1 = new Point(x,y);
        return p1;
    } // end method Point

    static void main (String[] args) {
        Point p1 = new Point(1,2);
        Point p2 = makePoint(3,4);
    } // end method main
} // end class Point

makePoint near end of constructor call
public class Point {
    int x;
    int y;
    Point (int xValue, int yValue) {
        x = xValue;
        y = yValue;
    } // end constructor

    Point makePoint (int x, int y) {
        Point p1 = new Point(x,y);
        return p1;
    } // end method Point

    static void main (String[] args) {
        Point p1 = new Point(1,2);
        Point p2 = makePoint(3,4);
    } end method main
} // end class Point
Methods are stacked

- When you call a method, its frame is *pushed* on the *call stack*
  - *Frame*: The ordered set of local variables that belong to a method
  - *Stack*: A list in which a data structure can be removed only if all objects added more recently than it are first removed. FILO.
  - A method stays on the stack until it ends (hits it closing curly brace).
  - If the foo() method calls the bar() method then the frame for bar() is stacked on top of foo(). When bar completes, it (and all its variables) are removed (poof!) and foo() continues.
- All local variables exist in on the stack, in the stack frame for the method call.
  - If the variable exists, it is *alive*
    - If another frame is on top of it, however, it is out of *scope*
  - If the method ends then the variable no longer exists in memory
- In Java, local variables get a default initialization of 0. Not true in C++!
Exception in thread “main”
java.lang.StringIndexOutOfBoundsException:
    String index out of range: 3
    at java.lang.String.charAt(Unknown Source)
    at Coordinate.getStatus (Coordinate: 67)
    at Ship.getLocation (Ship: 15)
    at GameBoard.checkTarget (GameBoard.java: 108)
    at GameBoard.play (GameBoard.java: 54)
    at Main.main (Main.java: 23)
Objects are allocated

- When you create an object instance memory is allocated on the heap.
  - There is NO WAY to create an object other than someone, somewhere using new on the class type
  - An object instance needs a place to store its instance variables.
  - The instance variables determine the size of the data structure.
  - *heap*: Essentially, a list in which data structures can be added or removed in any order and at any time.
  - Objects can be allocated anywhere in the heap where there is space (not necessarily adjacent to each other)
  - Any number of instances can be created, each starting at a unique address/location in the heap.
- Object reference variables (which could be static, local, or instance) hold the reference/location of the object on the heap
  - no live variable with reference \(<===>\) object is inaccessible garbage
Static variables have a permanent home

- When you create a static variable, it is assigned a permanent location in memory
  - A static variable is always live
    - (from the time the class is loaded at start until the program ends)
- Static variables don’t exist in the object’s data structure in the heap
  - This is why a static variables has the same value for all instances of a class. They all refer to the same memory location
- You should always assign an initial value to static variables
  - In Java and C++, static variables are assigned a default value of 0
null

- What value should an object reference variable have BEFORE an object is created and stores its reference in the variable?

- *null*
  - is a keyword
  - can be stored in a reference variable
  - Indicates that the variable current refers to *no existing object*

- The concept of null is particularly important in languages without garbage collection, as references may be stored to objects that have been destroyed

- In general, methods have a precondition that the parameters contain valid object references.
  - Best practices imply that one should always test for null references before using an object reference passed as a parameter
**Example: Memory**

```java
public class PointList extends Point {
    PointList nextPoint = new PointList ();
    static PointList firstPoint;

    PointList () {
        super(0,0);
    } // end constructor

    public static void main (String[] args) {
        PointList list = new PointList();
        list = null;
    } // end method main
} // end class Point
```

- What is the likely intention of this class?
- What are the static, instance, and local variables?
- This code compiles without error. When executed, you get a runtime exception: StackOverflowError. Why and how?
public class Deck {
    private ArrayList<Card> cardsInDeck;
    ...  
    public Card topCard() {
        return cardsInDeck.get(0);
    }  // end method topCard
}  // end class Deck

- Are cards private or not?
public class Poker {
    ...
    public void cheat () {
        Card topCard = topCard();
        topCard.setRank("Ace");
    } // end method cheat
} // end class Poker
Security and encapsulation

- When an accessor method exists to provide access to a private object:
  - Do not return the reference to the actual field object
    - This would allow the external program to access/change the object without going through the appropriate mutator method!
  - Instead, return a *copy* of the object.
    - This provides all the same information
    - But changes to the copy do *not* affect the original!

- You have to make a *new* object
- Wouldn’t it be great if we had a constructor that made this easy?
Copy constructors

- Copy constructor: A constructor that accepts an object of the same class as an argument and makes an “identical” copy.

```java
public class Deck {
    private ArrayList<Card> cardsInDeck;

    public Card topCard() {
        return new Card(cardsInDeck.get(0));
    } // end method topCard
} // end class Deck
```

```java
public class Card {
    public Card (Card originalCard) {
        this.setSuit(originalCard.getSuit());
        this.setRank(originalCard.getRank());
    } // end copy constructor

} // end class Card
```

Why are Strings Immutable in Java?
public class Deck {
    private ArrayList<Card> cardsInDeck;
    ...

    public Deck (Deck originalDeck) {
        this.cardsInDeck = new ArrayList<Card>();
        for (Card card : cardsInDeck) {
            this.cardsInDeck.add(card);
        }
    } // end (SHALLOW) copy constructor

} // end class Deck
public class Deck {
    private ArrayList<Card> cardsInDeck;
    ...

    public Deck (Deck originalDeck) {
        this.cardsInDeck = new ArrayList<Card>();
        for (Card card : cardsInDeck) {
            this.cardsInDeck.add(new Card(card));
        }
    } // end (SHALLOW) copy constructor

} // end class Deck

- To create a secure copy, you cannot forward any reference to any object (except immutable objects) in your aggregation
- Style: well designed objects should have a copy constructor that uses the copy constructors of the objects that it holds to ensure a deep copy