Inf1-OP

Classes and Objects - Part II

Volker Seeker, adapting earlier version by Perdita Stevens and Ewan Klein

School of Informatics

January 26, 2019

Objects ...

- have a static (compile-time) type defined inside a class
- are instances of classes created at runtime
- are created using a constructor and the new keyword
- are reference types

A homemade class: Circle

Next time, we'll see how to define a `Circle` class (in several variants). Let's start by seeing how we might use one.
Suppose its API is:

`public class Circle`

	`Circle(double radius)`	*constructor*
`double`	`getArea()`	
`void`	`enlarge(int scaleFactor)`	
`boolean`	`equals(Object o)`	*true iff o is a Circle of same size*

Using Circle

```
Circle c1 = new Circle(1);
double a1 = c1.getArea(); // pi

Circle c2 = new Circle(2);
double a2 = c2.getArea(); // 4 pi

Circle c3 = c1; // two references to same object
double a3 = c3.getArea(); // pi

System.out.println (c1 == c2); // false
System.out.println (c1.equals(c2)); // also false

System.out.println (c1 == c3); // true
System.out.println (c1.equals(c3)); // also true
```

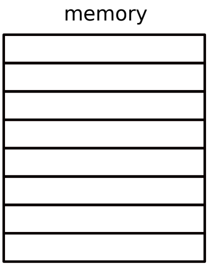
Using Circle, continued

```
c1.enlarge(2);

double a1new = c1.getArea(); // now 4 pi
double a2new = c2.getArea(); // still 4 pi
double a3new = c3.getArea(); // now 4 pi

System.out.println (c1 == c2); // still false
System.out.println (c1.equals(c2)); // now true

System.out.println (c1 == c3); // still true
System.out.println (c1.equals(c3)); // also still true
```

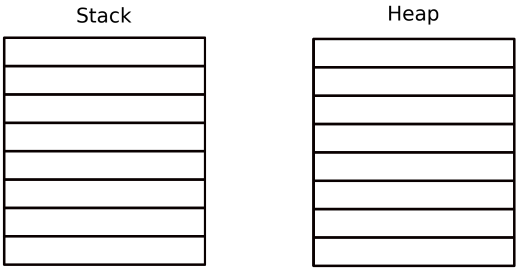
Stack vs. Heap

Where objects and local variables actually live!

Memory so far...



Actually more like this:



The Java Virtual Machine (JVM) manages memory in two different areas:

1. **The Stack**: for local variables
2. **The Heap**: for objects

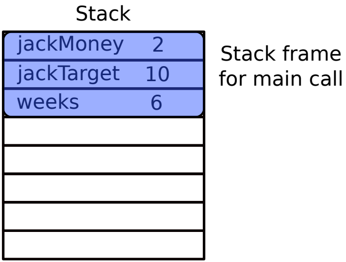
Stack Memory

Lets look at the Stack first.

```
public static int calcWeeks(int money, int target) {
  double sweets = 0.25;
  double redMoney = money * (1 - sweets);
  return (int) (target / redMoney);
}

public static String main(String[] args) {
  int jackMoney = 2;
  int jackTarget = 10;
  double weeks = calcWeeks(jackMoney, jackTarget);
}
```

Stack Memory

```
public static String
 main(String[] args) {
  int jackMoney = 2;
  int jackTarget = 10;
  double weeks =
    calcWeeks(jackMoney, jackTarget);
}
```



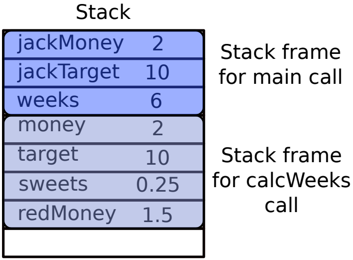
A little area on the stack, called a stack frame, is reserved for each function call. It holds:

- arguments given to the function
- local variables
- some extra stuff such as a return address to the caller

Lets ignore args for now.

Stack Memory

```
public static String
 main(String[] args) {
  int jackMoney = 2;
  int jackTarget = 10;
  double weeks =
   calcWeeks(jackMoney, jackTarget);
}
public static int
 calcWeeks(int money, int target) {
  double sweets = 0.25;
  double redMoney = money * (1 - sweets);
  return (int) (target / redMoney);
}
```



When a function call returns, its stack frame is removed from the stack and its return value copied into the caller's stack frame.

Recursion and Stack space

This knowledge can be important when working with recursive functions:

```
public int sumUp(int n) {
    if (n==1)
        return 1;
    else
       return sumUp(n-1)+n;
}
```

This program calculates the sum of all numbers from 1 until n.

Recursion and Stack space

This knowledge can be important when working with recursive functions:

```
public int sumUp(int n) {
    if (n==1)
        return 1;
    else
       return sumUp(n-1)+n;
}
```

This program calculates the sum of all numbers from 1 until n.

What can happen for very large n?

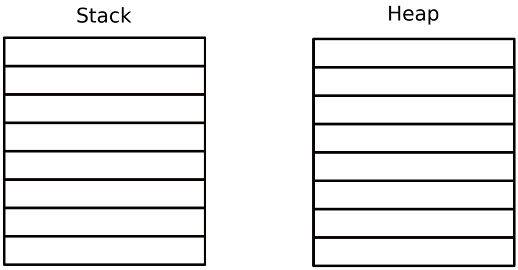
Recursion and Stack space

If a recursion is too deep, you can run out of stack memory.

java.lang.StackOverFlowError

Traditionally, the stack memory is much smaller than the heap memory. You can configure your JVM at program start time.

Stack and Heap

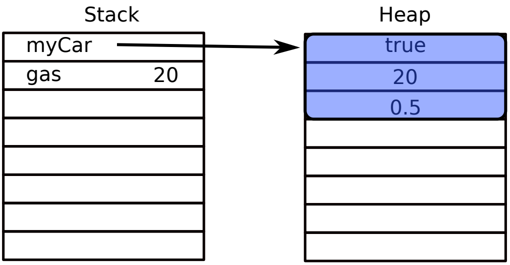


The Java Virtual Machine (JVM) manages memory in two different areas:

1. **The Stack**: for local variables
2. **The Heap**: for objects

Heap Memory

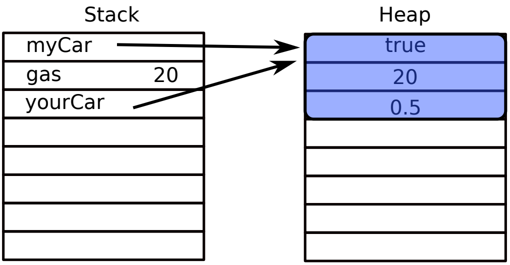
```
Car myCar = new Car();
myCar.startEngine();
int gas = 20;
myCar.accelerate(gas);
```



The memory of each object is put on the heap, while a reference to that object is kept on the stack.

Heap Memory

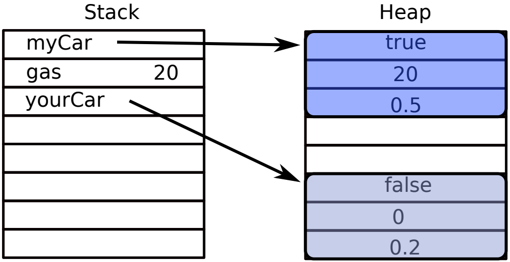
```
Car myCar = new Car();
myCar.startEngine();
int gas = 20;
myCar.accelerate(gas);
Car yourCar = myCar;
```



The memory of each object is put on the heap, while a reference to that object is kept on the stack.

Heap Memory

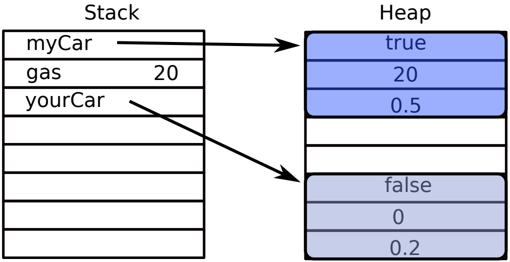
```
Car myCar = new Car();
myCar.startEngine();
int gas = 20;
myCar.accelerate(gas);
Car yourCar = new Car();
```



The memory of each object is put on the heap, while a reference to that object is kept on the stack.

Heap Memory

```
Car myCar = new Car();
myCar.startEngine();
int gas = 20;
myCar.accelerate(gas);
Car yourCar = new Car();
```

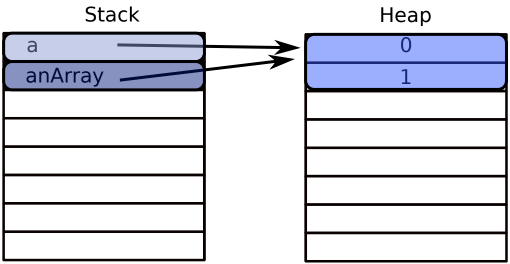


The memory of each object is put on the heap, while a reference to that object is kept on the stack.

Hence, passing objects around via function calls can lead to side effects.

Function call side effects

```
public static void
 addOne(int[] anArray){
  anArray[0]++;
}
public static void
 main(String[] args) {
  int[] a = { 0, 1 };
  addOne(a);
}
```



The array content on the heap is changed as a side effect of the function `AddOne`.

Lets practice that!

What does it print?

```java
public class AddOne {
    public static void addOne(int[] anArray) {
        anArray[0]++;
    }
    public static void main(String[] args) {
        int[] a = { 0, 1 };
        addOne(a);
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
```

What does it print?

```java
public class AddOne {
    public static void addOne(int[] anArray) {
        anArray[0]++;
    }
    public static void main(String[] args) {
        int[] a = { 0, 1 };
        addOne(a);
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
```

Prints **1 1**, due to call by reference and side effects.

What does it print?

```java
public class AddOne {
    public static void addOne(int[] anArray) {
        anArray = new int[2];
    }
    public static void main(String[] args) {
        int[] a = { 0, 1 };
        addOne(a);
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
```

What does it print?

```
public class AddOne {
    public static void addOne(int[] anArray) {
        anArray = new int[2];
    }
    public static void main(String[] args) {
        int[] a = { 0, 1 };
        addOne(a);
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
```

Prints **0 1**, since new memory is allocated in function.

What does it print?

```java
public class AddOne {
    public static int[] addOne(int[] anArray) {
        anArray = new int[2];
        return anArray;
    }
    public static void main(String[] args) {
        int[] a = { 0, 1 };
        a = addOne(a);
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
```

What does it print?

```
public class AddOne {
    public static int[] addOne(int[] anArray) {
        anArray = new int[2];
        return anArray;
    }
    public static void main(String[] args) {
        int[] a = { 0, 1 };
        a = addOne(a);
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
```

Prints **0 0**, since new memory is allocated, automatically initialised and returned to replace the original array reference in main.

Immutuability

Side effects can be dangerous. You can take precautions by using immutuables.

An **immutuable** object cannot change its state after it has been created, e.g. String, Integer, etc.

Circle and Car are **mutuable**.

Immutuability

Side effects can be dangerous. You can take precautions by using immutuables.

An **immutuable** object cannot change its state after it has been created, e.g. String, Integer, etc.

Circle and Car are **mutuable**.

It allows other fancy things such as interning and copying the object by simply copying the references.

Immutuability

Side effects can be dangerous. You can take precautions by using immutuables.

An **immutuable** object cannot change its state after it has been created, e.g. String, Integer, etc.

Circle and Car are **mutuable**.

It allows other fancy things such as interning and copying the object by simply copying the references.

This is what you would call a **Design Pattern**.

Objects ...

- have a static (compile-time) type defined inside a class
- are instances of classes created at runtime
- are created using a constructor and the new keyword
- are reference types
- reside on the heap memory rather than the stack

Immutuability

```java
public class AddOne {
    public static void addOne(int[] anArray) {
        anArray = new int[2];
    }
    public static void main(String[] args) {
        int[] a = { 0, 1 };
        addOne(a);
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
```

A new array is allocated on the Heap in function `AddOne`.

What happens to its memory when the function returns?

Cleaning up the Heap

If objects on the Heap are no longer referenced by anyone, an automatic process called **garbage collection** cleans it up.

Without the cleanup, the `AddOne` function would **leak** memory every time it is called until:

java.lang.OutOfMemoryError: Java Heap Space

Referencing nothing

We have talked about cleaning up the Heap after all references are gone. What happens in the opposite case?

Where do references point when there is no corresponding object on the heap?

Referencing nothing

We have talked about cleaning up the Heap after all references are gone. What happens in the opposite case?

Where do references point when there is no corresponding object on the heap?

```
Car myCar = null;
```

The null literal indicates an object reference pointing at nothing.

Using the `myCar` variable to call a method on it or change its state will now result in a **java.lang.NullPointerException**.

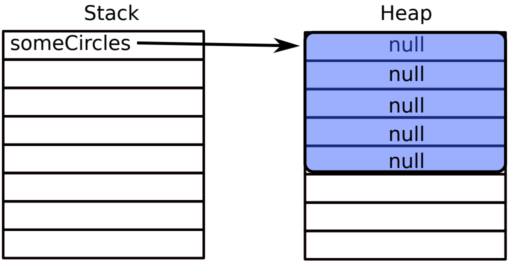
Null - Know the difference!

Objects containing Objects

```
// Allocate space for 5 refs to Circles:
Circle[] someCircles = new Circle[5];
```

Objects containing Objects

```
// Allocate space for 5 refs to Circles:
Circle[] someCircles = new Circle[5];
```

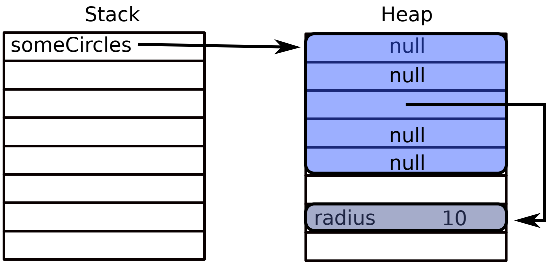


An array of objects is automatically initialised with null.

To fill it, space for each object needs to be allocated explicitely.

Objects containing Objects

```
// Allocate space for 5 refs to Circles:
Circle[] someCircles = new Circle[5];

someCircles[2] = new Circle(10);
```

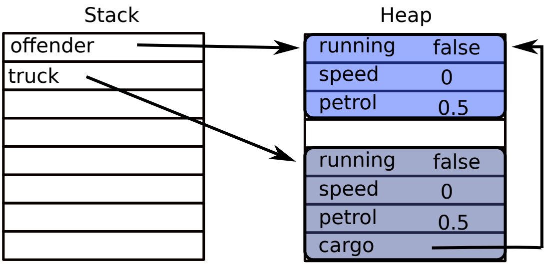


An array of objects is automatically initialised with null.

To fill it, space for each object needs to be allocated explicitely.

Objects containing Objects

```
Car offender = new Car();
TowTruck truck = new TowTruck(offender);
```



The same is true for class instances containing other class instances.

Shallow vs. Deep Copy

```
Circle[] someCircles = new Circle[5];
for(int i = 0; i < someCircles.length; i++)
    someCircles[i] = new Circle(i * 10);

Circle[] shallowCopy = new Circle[5];
for(int i = 0; i < shallowCopy.length; i++)
    shallowCopy[i] = someCircles[i];

Circle[] deepCopy = new Circle[5];
for(int i = 0; i < deepCopy.length; i++)
    deepCopy[i] = new Circle(someCircle[i].radius);
```

Careful when copying objects containing objects:

- **shallow copy** copies only the references of the contained objects
- **deep copy** also creates new memory for the contained objects and copies the state

Objects ...

- have a static (compile-time) type defined inside a class
- are instances of classes created at runtime
- are created using a constructor and the new keyword
- are reference types
- reside on the heap memory rather than the stack
- are destroyed automatically by the garbage collector

Summary

- JVM manages memory in two different areas
  - Stack: for local variables
  - Heap: for objects
- Watch out with recursion and function side effects
- An object variable containing null references no memory
- Stack frames are cleaned once the function scope is left
- Garbage collection cleans up the heap
- Immutuable objects cannot change their state once initialised
- Watch out with Deep Copy vs Shallow Copy

Reading

Java Tutorial

Reading on objects and classes will follow once I finish it next week.

Blog article about Heap and Stack:

https://www.journaldev.com/4098/java-heap-space-vs-stack-memory