Memory

Brahm Capoor
int x = 42;
int y = 42;

if (x == y) {
    println("These numbers are equal");
} else {
    println("These numbers are different");
}
int x = 42;
int y = 42;

if (x == y) {
    println("These numbers are equal");
} else {
    println("These numbers are different");
}
GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);

if (r1 == r2) {
    println("These rectangles are equal");
} else {
    println("These rectangles are different");
}
GRect r1 = new GRect(20, 50);  
GRect r2 = new GRect(20, 50);  

if (r1 == r2) {
    println("These rectangles are equal");
} else {
    println("These rectangles are different");
}
An intuition: You can have different rectangles with the same properties, but you can’t have different 42s
A hot take: being the same thing is different from having the same properties
A hot take: identity and equality are different
A hot take: identity and equality are different

The same thing

The same properties
A hot take: **identity** and **equality** are different

The same thing

The same properties
What do we know about variables?

```
public void run() {
    int x = 42;
    int y = 2;
    int z = foo(x, y);
}

private void foo(int a, int b) {
    int x = a + 2 * b;
    return x + 3;
}
```

Every time a method is called, we make a new stack frame for it.
What do we know about variables?

```java
public void run() {
    int x = 42;
    int y = 2;
    int z = foo(x, y);
}

private void foo(int a, int b) {
    int x = a + 2 * b;
    return x + 3;
}
```

Every time a variable is created, we make a new box for it.
What do we know about variables?

Every time a variable is created, we make a new box for it
What do we know about variables?

```java
public void run() {
    int x = 42;
    int y = 2;
    int z = foo(x, y);
}

private void foo(int a, int b) {
    int x = a + 2 * b;
    return x + 3;
}
```

We evaluate the right-hand side **before** the left-hand side.
What do we know about variables?

Every time a method is called, we make a new stack frame for it and copy parameter values.

```java
public void run() {
    int x = 42;
    int y = 2;
    int z = foo(x, y);
}

private void foo(int a, int b) {
    int x = a + 2 * b;
    return x + 3;
}
```
What do we know about variables?

```
public void run() {
    int x = 42;
    int y = 2;
    int z = foo(x, y);
}

private void foo(int a, int b) {
    int x = a + 2 * b;
    return x + 3;
}
```

Every time a variable is created, we make a box for it.
What do we know about variables?

public void run() {
    int x = 42;
    int y = 2;
    int z = foo(x, y);
}

private void foo(int a, int b) {
    int x = a + 2 * b;
    return x + 3;
}

Returning allows a method to pass information back to its caller.
What do we know about variables?

public void run() {
    int x = 42;
    int y = 2;
    int z = foo(x, y);
}

private void foo(int a, int b) {
    int x = a + 2 * b;
    return x + 3;
}

When a method returns, its stack frame gets erased.
Let’s talk about the boxes

The boxes for variables are stored in our computer’s memory
Let’s talk about the boxes

The boxes for variables are stored in our computer’s memory.

These boxes are a fixed size, and just large enough to store an int and other primitive variables.
Let’s talk about the boxes

The boxes for variables are stored in our computer’s memory.

These boxes are a fixed size, and just large enough to store an int and other primitive variables.

All of these boxes are in a part of memory called the Stack.
What happens when we make an object?

```java
GRect rect = new GRect(40, 50);
```
What happens when we make an object?

```java
GRect rect = new GRect(40, 50);
```

Ask the GRect factory for a new GRect
What happens when we make an object?

```java
GRect rect = new GRect(40, 50);
```

Now where should we put it?
What happens when we make an object?

What makes this a hard problem?

Primitive variables (like ints) contain less information than Objects (like GRects) and so can fit in boxes on the Stack.
What happens when we make an object?

What makes this a hard problem?

Primitive variables (like `ints`) contain less information than `Objects` (like `GRects`) and so can fit in boxes on the Stack.

The only information a primitive variable represents is its `value`.
What happens when we make an object?

What makes this a hard problem?

Primitive variables (like ints) contain less information than Objects (like GRects) and so can fit in boxes on the Stack.

The only information a primitive variable represents is its value.

An object like a GRect packages together lots of smaller pieces of information, like dimensions, location and color.
What happens when we make an object?

What makes this a hard problem?

Primitive variables (like `ints`) contain less information than `Objects` (like `GRects`) and so can fit in boxes on the Stack

The only information a primitive variable represents is its value

An object like a `GRect` packages together lots of smaller pieces of information, like dimensions, location and color

The place in memory to store larger pieces of information like this is the Heap
What happens when we make an object?

```
GRect rect = new GRect(40, 50);
```

We put the new GRect in the Heap!
What happens when we make an object?

```java
GRect rect = new GRect(40, 50);
```

Now, we need to know where to find this `GRect` in the Heap.
What happens when we make an object?

GRect rect = new GRect(40, 50);

So we store that information on the Stack!

The Heap
What happens when we make an object?

```java
GRect rect = new GRect(40, 50);
```

So we store that information on the Stack!
What happens when we make an object?

```java
GRect rect = new GRect(40, 50);
```

More specifically, we store the address of the new `GRect` on the Stack.
What happens when we make an object?

```
GRect rect = new GRect(40, 50);
```

You can think of the stack variable as a URL...

...and the object on the heap as a website
Our takeaways

Primitive variables are stored on the Stack

```java
int x = 42;
```

This value is on the Stack
Our takeaways

Primitive variables are stored on the **Stack**

```java
int x = 42;
```

This value is on the **Stack**

Objects are stored on the **Heap** and referred to from the **Stack**

```java
GRect r = new GRect(...);
```

This reference to an object is on the **Stack**

This object is on the **Heap**
Our takeaways

**Primitive variables are stored on the Stack**

```java
int x = 42;
```

**Objects are stored on the Heap and referred to from the Stack**

```java
GRect r = new GRect(...);
```

- This value is on the Stack
- This reference to an object is on the Stack
- This object is on the Heap
public void run() {
    GRect first = new GRect(20, 10);
    GRect second = first;
    second.setColor(Color.GREEN);
    add(first, 0, 0);
}

First, we make a new GRect on the Heap and store a reference to it on the Stack.
An example

```java
public void run() {
    GRect first = new GRect(20, 10);
    GRect second = first;
    second.setColor(Color.GREEN);
    add(first, 0, 0);
}
```

Next, we copy that reference to another spot on the stack.
An example

```java
public void run() {
    GRect first = new GRect(20, 10);
    GRect second = first;
    second.setColor(Color.GREEN);
    add(first, 0, 0);
}
```

Then, we call a method on the object that we’re referencing.
An example

```java
public void run() {
    GRect first = new GRect(20, 10);
    GRect second = first;
    second.setColor(Color.GREEN);
    add(first, 0, 0);
}
```

Finally, we pass that object into the `add()` method.
Now for a quick detour
What’s in a box?

We saw earlier that each ‘box’ on the stack has a fixed size.
What’s in a box?

We saw earlier that each ‘box’ on the stack has a fixed size.

That fixed size is called a word.
What’s in a box?

We saw earlier that each ‘box’ on the stack has a fixed size.

That fixed size is called a word.

On most computers today, a word is 64 bits (1s and 0s).
What’s in a box?

We saw earlier that each ‘box’ on the stack has a **fixed size**

That fixed size is called a **word**

On most computers today, a word is 64 **bits** (1s and 0s)

**8 bits are called a byte**
What’s in a name?

Bit is short for **Binary Digit** (a 1 or a 0)
What’s in a name?

Bit is short for **Binary Digit** (a 1 or a 0)

... but Bytes were given that name because of the ~wordplay~
What’s in a name?

Bit is short for Binary Digit (a 1 or a 0)

... but Bytes were given that name because of the ~wordplay~

... and half a byte is called a Nybble!
What’s in a name?

Bit is short for **Binary Digit** (a 1 or a 0)

... but Bytes were given that name because of the ~**wordplay**~

... and half a byte is called a **Nybble**!
// end of detour
Modifying primitive parameters

```java
double x = 42;
foo(x);
println(x);
```

```java
public void foo(int x) {
    x = 7;
}
```

Every time a variable is created, we make a new box for it.
Modifying primitive parameters

```java
public void run() {
    int x = 42;
    foo(x);
    println(x);
}
```

```java
public void foo(int x) {
    x = 7;
}
```

Every time a method is called, we make a new stack frame for it and copy the parameter values in the Stack.
Modifying primitive parameters

When we modify a variable, we change the box in the current stack frame.
Modifying primitive parameters

```java
public void run() {
    int x = 42;
    foo(x);
    println(x);
}

public void foo(int x) {
    x = 7;
}
```

When we modify a variable, we change the box in the current stack frame.
Modifying primitive parameters

```java
public void run() {
    int x = 42;
    foo(x);
    println(x);
}

public void foo(int x) {
    x = 7;
}
```

When we modify a variable, we change the box in the current stack frame.
Modifying primitive parameters

```java
public void run() {
    int x = 42;
    foo(x);
    println(x); // prints 42
}

public void foo(int x) {
    x = 7;
}
```

When a method returns, its stack frame gets erased
Pass by copy

When we pass primitives as parameters, we pass copies of their values.
Modifying object parameters

```java
public void run() {
    GRect r = new GRect(5, 5);
    foo(r);
    add(r);
}

public void foo(GRect r) {
    r.setColor(Color.GREEN);
}
```

When we make an object, we store a reference to the Heap on the Stack.
public void run() {
    GRect r = new GRect(5, 5);
    foo(r);
    add(r);
}

public void foo(GRect r) {
    r.setColor(Color.GREEN);
}
Modifying object parameters

```java
public void run() {
    GRect r = new GRect(5, 5);
    foo(r);
    add(r);
}

public void foo(GRect r) {
    r.setColor(Color.GREEN);
}
```

Every time a method is called, we make a new stack frame for it and copy the parameter values in the Stack.
Modifying object parameters

```java
public void run() {
    GRect r = new GRect(5, 5);
    foo(r);
    add(r);
}

public void foo(GRect r) {
    r.setColor(Color.GREEN);
}
```

Every time a method is called, we make a new stack frame for it and copy the parameter values in the Stack.
Modifying object parameters

```java
public void run() {
    GRect r = new GRect(5, 5);
    foo(r);
    add(r);
}

public void foo(GRect r) {
    r.setColor(Color.GREEN);
}
```

Every time a method is called, we make a new stack frame for it and copy the parameter values in the Stack.
Modifying object parameters

```java
public void run() {
    GRect r = new GRect(5, 5);
    foo(r);
    add(r);
}

public void foo(GRect r) {
    r.setColor(Color.GREEN);
}
```

When a method returns, its stack frame gets **erased**.
Modifying object parameters

```java
public void run() {
    GRect r = new GRect(5, 5);
    foo(r);
    add(r); // green rectangle
}

public void foo(GRect r) {
    r.setColor(Color.GREEN);
}
```

The changes persisted!
Pass by reference

When we pass objects as parameters, we pass copies of references to the same object.
A general heuristic

There are two kinds of variables on the Stack: Primitive values and Object references
A general heuristic

There are two kinds of variables on the Stack: Primitive values and Object references.

When we pass, copy or return something, we’re modifying the Stack.
A general heuristic

There are two kinds of variables on the Stack: Primitive values and Object references.

When we pass, copy or return something, we’re modifying the Stack.

When we call a method on an Object reference, we follow the reference to the Heap.
A general heuristic

There are two kinds of variables on the Stack: Primitive values and Object references.

When we pass, copy or return something, we’re modifying the Stack.

When we call a method on an Object reference, we follow the reference to the Heap.

We can have multiple references to the same object.
A general heuristic

There are two kinds of variables on the Stack: Primitive values and Object references.

When we pass, copy or return something, we’re modifying the Stack.

When we call a method on an Object reference, we follow the reference to the Heap.

We can have multiple references to the same object:
- Modifying the object changes it for all of its references.
A general heuristic

There are two kinds of variables on the Stack: Primitive values and Object references.

When we pass, copy or return something, we’re modifying the Stack.

When we call a method on an Object reference, we follow the reference to the Heap.

We can have multiple references to the same object:
- Modifying the object changes it for all of its references.
- Modifying a reference changes only what it’s pointing at.
A general heuristic

There are two kinds of variables on the Stack: Primitive values and Object references.

When we pass, copy or return something, we’re modifying the Stack.

When we call a method on an Object reference, we follow the reference to the Heap.

We can have multiple references to the same object:

- Modifying the object changes it for all of its references.
- Modifying a reference changes only what it’s pointing at.
A hot take: **identity** and **equality** are different

- The same thing
- The same properties
A hot take: identity and equality are different

```java
int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}
```
A hot take:
identity and equality are different

```java
int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);

if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}
```
A hot take: identity and equality are different

```java
int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}
```

```java
GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}
```

The values of these primitives on the Stack are equal
A hot take: identity and equality are different.

```java
int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}
```

The values of these primitives on the Stack are equal.

r1 and r2 reference different objects on the Heap.
A hot take: identity and equality are different.

```java
int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}
```

The values of these primitives on the Stack are equal.

The value of these references on the Stack are different.
A hot take: identity and equality are different.

```java
int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}
```

The values of these primitives on the Stack are equal.
The value of these references on the Stack are different.
```java
int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}
```

**identity**

The `==` operator compares whatever values are on the Stack.

**equality**

The values of these primitives on the Stack are equal.

The value of these references on the Stack are different.
```java
int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}

A hot take:
identity and equality are different

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}

int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}
```

*identity*

The `==` operator compares whatever values are on the Stack.

The values of these primitives on the Stack are equal.

*equality*

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}
int x = 42;
int y = 42;
if (x == y) {
  println("Equal");
} else {
  println("Different");
}

A hot take:
identity and equality are different

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
  println("Equal");
} else {
  println("Different");
}

The values of these primitives on the Stack are equal

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
  println("Equal");
} else {
  println("Different");
}

The == operator compares whatever values are on the Stack

The values of these references on the Stack are different

We want to compare what r1 and r2 are referencing on the heap
int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}

The == operator compares whatever values are on the Stack
The values of these primitives on the Stack are equal

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1.equals(r2)) {
    println("Equal");
} else {
    println("Different");
}

We want to compare what r1 and r2 are referencing on the heap
The value of these references on the Stack are different
int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}

The value of these primitives on the Stack are equal

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1.equals(r2)) {
    println("Equal");
} else {
    println("Different");
}

We want to compare what r1 and r2 are referencing on the heap

The Stack
- r1
  - 42
- r2
  - 60

The Heap
- memory.com/42
- memory.com/60

The value of these references on the Stack are different

The == operator compares whatever values are on the Stack

A hot take:
- identity and equality are different
int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}

The values of these primitives on the Stack are equal

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1.equals(r2)) {
    println("Equal");
} else {
    println("Different");
}

We want to compare what r1 and r2 are referencing on the heap

The Stack
r1 42
r2 60

The Heap
memory.com/42
memory.com/60

== compares the values in the Stack

identity

equality
int x = 42;
int y = 42;
if (x == y) {
    println("Equal");
} else {
    println("Different");
}

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1 == r2) {
    println("Equal");
} else {
    println("Different");
}

The == operator compares whatever values are on the Stack

The values of these primitives on the Stack are equal

GRect r1 = new GRect(20, 50);
GRect r2 = new GRect(20, 50);
if (r1.equals(r2)) {
    println("Equal");
} else {
    println("Different");
}

We want to compare what r1 and r2 are referencing on the heap

The == compares the values in the Stack
.equals() compares objects on the Heap
identity

Compared using the `==` operator

equality

Compared using the `.equals()` method
identity

Compared using the `==` operator

Compares values and references on the Stack

equality

Compared using the `.equals()` method

Follows references to compare objects on the Heap
identity

Compared using the `==` operator

Compares values and references on the Stack

Applies to objects and primitives

equality

Compared using the `.equals()` method

Follows references to compare objects on the Heap

Applies only to objects
Boxes on the Stack are a fixed size.

An object’s identity is separate from its properties.
Boxes on the Stack are a fixed size

We store objects on the Heap and refer to them from the Stack

An object’s identity is separate from its properties
Boxes on the Stack are a fixed size.

We store objects on the Heap and refer to them from the Stack.

We use references to efficiently transfer large objects.

An object’s identity is separate from its properties.
Boxes on the Stack are a fixed size.

We store objects on the Heap and refer to them from the Stack.

Primitives are passed by copy and Objects are passed by reference.

We use references to efficiently transfer large objects.

An object’s identity is separate from its properties.
Boxes on the Stack are a fixed size.

We store objects on the Heap and refer to them from the Stack.

Primitives are passed by copy and Objects are passed by reference.

We use references to efficiently transfer large objects.

An object’s identity is separate from its properties.

We can distinguish between equality and identity!
Boxes on the Stack are a fixed size

We store objects on the Heap and refer to them from the Stack

Primitives are passed by copy and Objects are passed by reference

We use references to efficiently transfer large objects

An object’s identity is separate from its properties

We can distinguish between equality and identity!
The Burroughs B5000 computer, 1961