Introduction to OOP

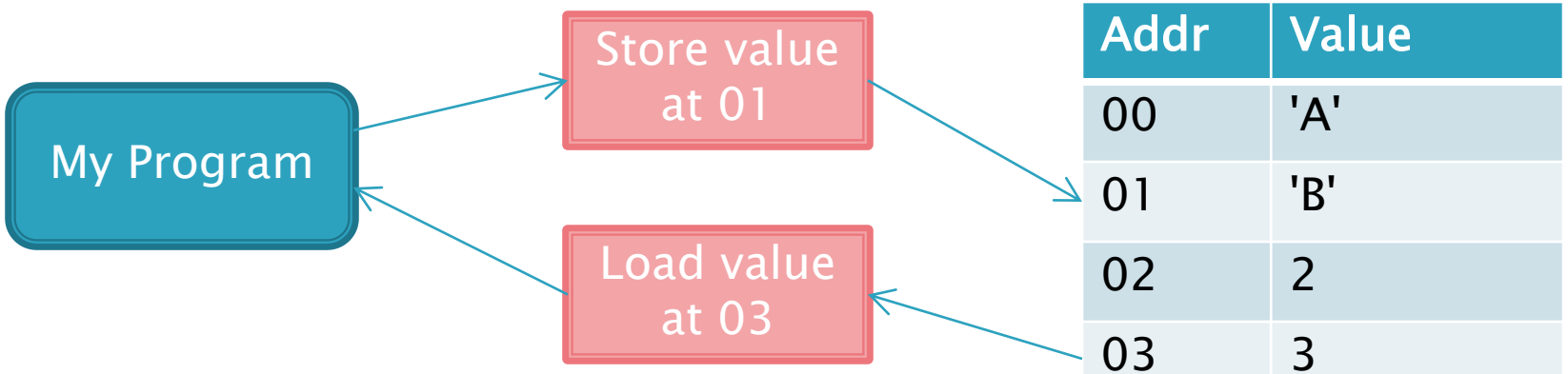
Object Oriented Programming in Java

EHSDI

eBuzima

In the Beginning…

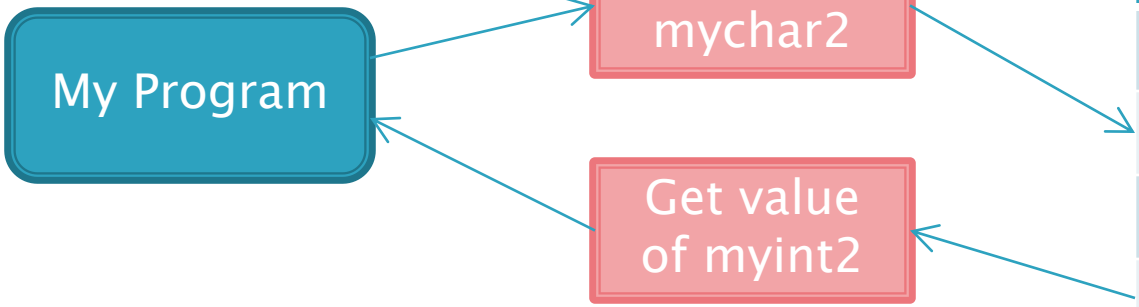
- There was just memory!
- If you wanted to store a value, you had to pick a memory address



Addr	Value
00	'A'
01	'B'
02	2
03	3

Then there were variables

- A variable allowed the programmer to give a piece of memory a name, which the compiler would remember



Variable	Value
mychar1	'A'
mychar2	'B'
myint1	2
myint2	3

But what about complex data…

- Variables only worked for simple data types like integers and characters
- For example a coordinate consists of an x value and a y value. With just variables we would have to create two variables for each coordinate

```
int myCoord1_X = 2
int myCoord1_Y = 4

int myCoord2_X = 8
int myCoord2_Y = -2
```

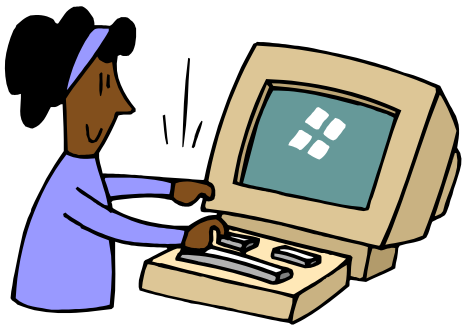
Then there were structures

- A structure (as it was called in C) allowed simple values to be grouped into more complex data types
- So we could define a new data type – like a Coordinate data type…

```
struct Coordinate {
   int x
   int y
}

Coordinate c1 = { 2, 4 }
Coordinate c2 = { 8, -2 }
```

Working with structures

- Typically a programmer would create a set of functions which operated on a given structure



```
struct Coordinate {
  int x
  int y
}

coordinate_draw(Coordinate c) {
  // Code to draw a coordinate
}

coordinate_reset(Coordinate c) {
  // Code to draw a coordinate
}
```

Member functions (methods)

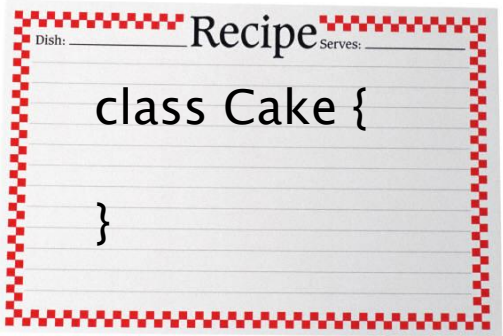
- Eventually someone realized that life would be easier if structures could have their own functions…

- And the called it a ***class!***

```
class Coordinate {
  int x
  int y

  draw() {
    // Code to draw a coordinate
  }

  reset() {
    // Code to draw a coordinate
  }
}
```

Classes vs. Objects

- A class is like a blueprint or a recipe for creating objects. It tells the JVM how to make an object (an instance) of that particular type



The Class

The JVM

Objects

Constructors

- Sometimes we need to given the JVM some extra instructions on how to create an instance of a class
- A constructor is a method which has the same name as the class, and no return type
- We can use it to initialize data members…

```
class Coordinate {
 int x, y;

 Coordinate() {
  x = 0;
  y = 0;
 }
}
```

Constructors: Default vs. Explicit

- A *default* constructor is one that has no parameters
- An *explicit* constructor is one that takes parameters, and these are usually used to initialize the data members

```
class Coordinate {
 int x, y;

 // Default
 Coordinate() {
  this.x = 0;
  this.y = 0;
 }

 // Explicit
 Coordinate(int x, int y) {
  this.x = x;
  this.y = y;
 }
}
```

Constructors: Implicit

- If you don't specify any constructors in your class, the compiler will create an *implicit default* constructor automatically
- It's just an empty default constructor

```
class Coordinate {
 int x, y;

 Coordinate() {
 }

}
```

Something new…

- **new** is a Java keyword which tells the JVM to create an new object of a given class type

```
Coordinate c = new Coordinate(2, 4);
String s = new String("Hello");
Integer i = new Integer(3);
```

- It's always followed by a constructor (default, explicit or implicit), which it calls to create the object

Reference vs Object

- When we create an object, we usually assign a reference to it to a variable, i.e.

```
Coordinate c = new Coordinate(2, 4);
```

- `c` is a variable – a reference to the object which now exists in memory. We can assign it a new different object…

```
c = new Coordinate(8, -2);
```

- and then the first one will be lost!

This is a keyword

- The `this` keyword is used within methods to access the current class
- It can be used to differentiate between instance variables and method parameters that have the same name, e.g.

```
class Shape {
  int color;
  void setColor(int color) {
    this.color = color;
  }
}
```

Duplication = Wasted time

- Sometimes a class would require some of the same variables and methods as another class
- Duplicating code is ALWAYS bad
- Programmers needed a way to share members between different classes, so someone invented *inheritance*

Duplication = Wasted time

- Supposing we created a set of classes for different shapes. Much of our code might be duplicated…

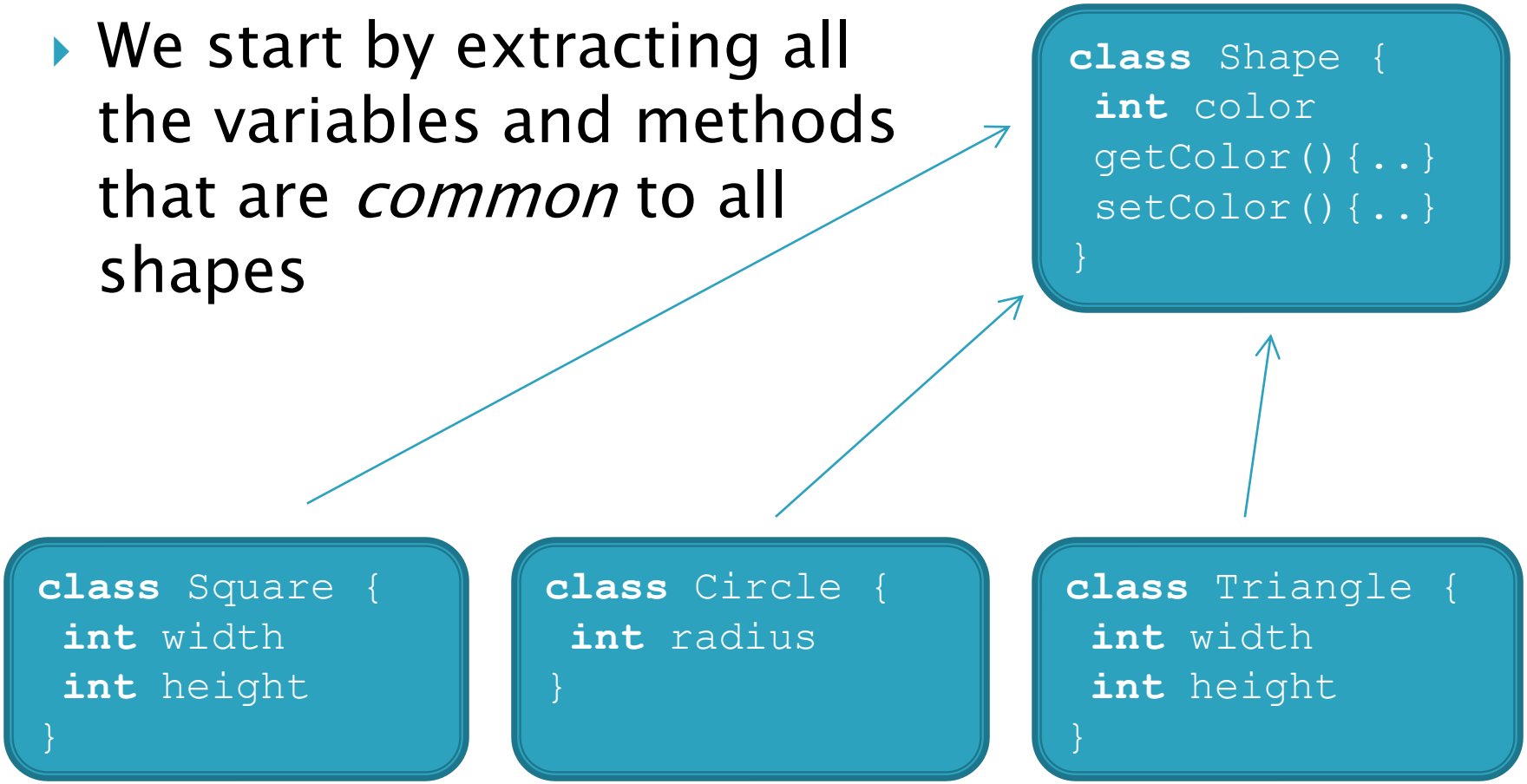
```
class Square {
 int color
 int width
 int height

 getColor(){..}
 setColor(){..}
}
```

```
class Circle {
 int color
 int radius

 getColor(){..}
 setColor(){..}
}
```

```
class Triangle {
 int color
 int width
 int height

 getColor(){..}
 setColor(){..}
}
```

Inheritance to the rescue!

- We start by extracting all the variables and methods that are *common* to all shapes



Extending a class

- Then we use the **extends** keyword (in Java) to tell the compiler that *Square*, *Circle* and *Triangle* should inherit all the functionality from *Shape*

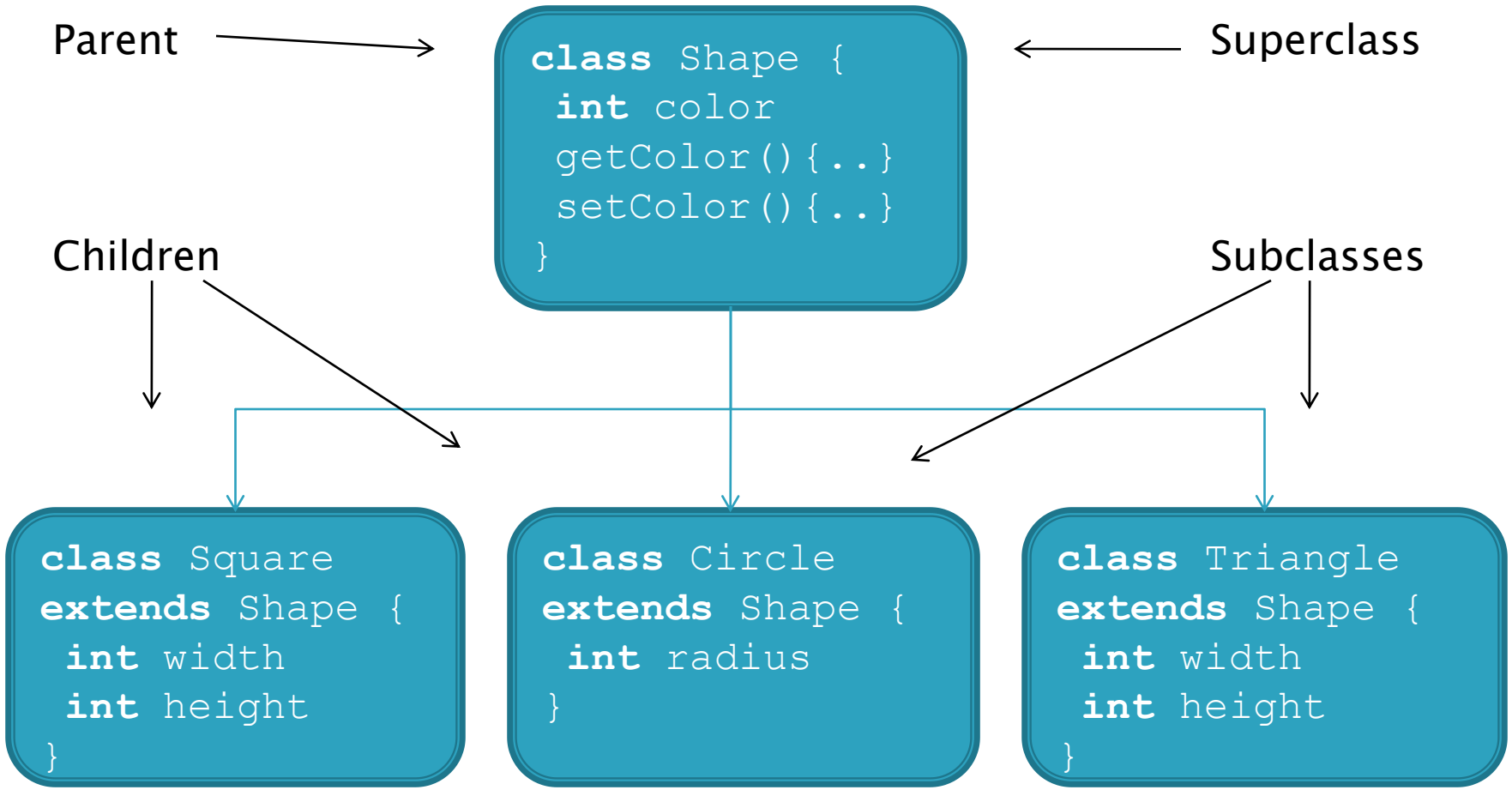
```
class Shape {
 int color
 getColor(){..}
 setColor(){..}
}
```

```
class Square
extends Shape {
 int width
 int height
}
```

```
class Circle
extends Shape {
 int radius
}
```

```
class Triangle
extends Shape {
 int width
 int height
}
```

A Family Tree



Circle *is a* Shape

- `Circle` doesn't just import functionality from `Shape`, `Circle` is a `Shape`, so we can do this…

```
Shape c = new Circle();
int i = c.getColor();
```

- The object created by the JVM is of type `Circle`, but we can reference it with variable of type `Shape`

But Shape *is not a* Circle!

- Inheritance goes one way – `Shape` is not a kind of `Circle`, so this won't compile…

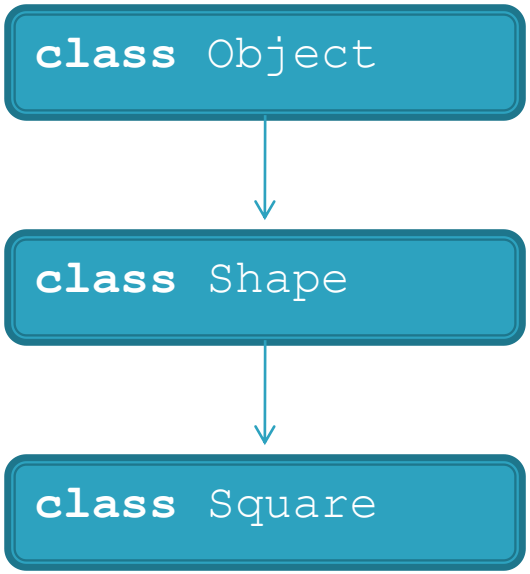
```
Circle c = new Shape();  // Error!!!!
```

- And this definitely won't work (Circle and Triangle are siblings)

```
Circle c = new Triangle();  // Error!!!!
```

Every object is an Object…

- Anytime you create a class in Java, it automatically inherits from a class called Object
- So every class is related!




```
Object o = new Shape(); // Works!!
```

Casting Primitives…

- *Casting* means converting from one data type to another. We can do it with primitive types…

```
byte b = 34;
int n = b;          // Implicit cast
int m = (int)b;     // Explicit cast
byte c = (byte)n;   // Explicit cast
byte d = n;         // Error !!!
```

Casting Objects…

- We can also cast objects as long as the object being cast ***is*** what we're casting it to!

```
Square q = new Square();

Shape s = q;          // Implicit cast
Shape t = (Shape)q;   // Explicit cast
```

- The above code works because `Square` ***is a*** `Shape`

Casting Objects…

- But what if the reference type isn't exactly the same as the object type?
- The compiler will throw an error if you try an implicit cast
- But the JVM will do the cast, as long as the actual object is of the right type…

```
Shape s = new Square();
Object o = new Square();

Square t = s;             // Compiler Error
Square q = (Square)s;     // Works!!!
Square r = (Square)o;     // Works!!!
```

Public and Private

- We can control who can access the instance variables and methods of our classes
- **public** means that all other classes can access the variable/method
- **private** means that only the class itself can access the variable/method

```
class Shape {
  public int color;
  private int opacity;
}
```

Protected

- **protected** means that the members are *only* accessible within the class and within all subclasses

```
class Shape {
  protected int color;  // Declared protected
}

class Square extends Shape {
  // color is accessible here
}
```

Package-private (the default)

- If we don't specify an access modifier explicitly, then that member is *package-private*

```
package intro;

class Shape {
  int color;  // Will be package-private
}

class AnotherClass {
  // color is accessible here because we're in the
  // same package
}
```

Overriding Methods and Variables

- When you extend a class, to create a subclass, you can override some of its methods or variables
- An overridden method has the same name, parameters and return type as the method it is overridding
- When you call the method on an instance of the subclass, even if the reference is of the superclass type, the JVM will execute the overridden method in the subclass

Overriding Methods and Variables

- For example….

```
class Shape {
  public void Print() { System.out.println("Shape");}
}
class Square extends Shape {
  public void Print() {System.out.println("Square");}
}
```

```
Square q = new Square();
Shape s = (Shape)q;
q.Print();
s.Print();
```

Outputs…

```
Square
Square
```

Preventing overriding with final

- You can prevent subclasses from overriding methods or variables using the **final** keyword

```
class Shape {
  public final void Print() {
    System.out.println("Shape");
  }
}
class Square extends Shape {
  public void Print() {…}        // Compiler error!
}
```

That's super!

- The **super** keyword is used to access the members of the superclass from a subclass
- It can be used within a method to call a method in the superclass…

```
class Shape {
  public void Print() { System.out.println("Shape");}
}
class Square extends Shape {
  public void Print() {
    super.Print();
    System.out.println("Square");
  }
```

That's super!

- Or it can be used to pass parameters to a constructor of the superclass

```
class Shape {
  Shape(int color) {…}
}
class Square extends Shape {
  Square(int color) {
    super(color);
  }
}
```

- `Square` has to define a constructor which matches `Shape`'s