# Introduction to Java (cs2514)

## Lecture 3 & 4: Classes and Objects

M. R. C. van Dongen

January 22, 2018

# Objects and Classes

- Programmers construct their Java program from *objects*.
- Similar to a builder building a house from parts:
  - Doors;
  - Windows;
  - Walls;
  - ...
- Each part has its own function.
- The parts work together to form the house:
  - The house is the *sum* of the parts.
- The builder doesn't have to construct the parts.
- All he does is composing them.

# Using Objects

- Objects are the first citizens of Java programs.
- You make an object work by calling its methods.
- Each method is a sequence of instructions.
- You can call a method even if you don't know its instructions.

**Java**

```
System.out.println( "Hello world!" );
```

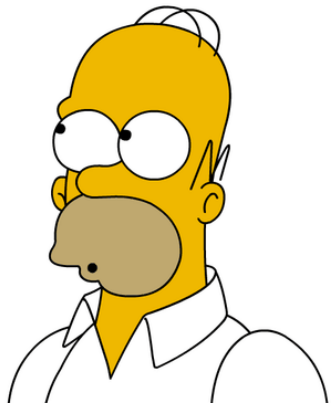
- Each method provides a service.
  - The method performs the service when you call the method.
- Different methods may provide different services:
  - Draw a picture;
  - Print text;
  - Set up a connection with another computer;
  - Compute something and return it;
  - ...

# Classes

- Each object belongs to a unique class.
- Different objects may belong to different classes.
  - `System.out`
  - `"Hello world!"`
- An object that belongs to a class is called an *instance* of the class.
- A class may have more than one instance:
  - `"Hello world!"`
  - `"What's up Doc?"`
  - ...

# Classes

- Each object belongs to a unique class.
- Different objects may belong to different classes.
  - `System.out`
  - `"Hello world!"`
- An object that belongs to a class is called an *instance* of the class.
- A class may have more than one instance:
  - `"Hello world!"`
  - `"What's up Doc?"`
  - ...



# Classes

- Each object belongs to a unique class.
- Different objects may belong to different classes.
  - `System.out`
  - `"Hello world!"`
- An object that belongs to a class is called an *instance* of the class.
- A class may have more than one instance:
  - `"Hello world!"`
  - `"What's up Doc?"`
  - ...

# Classes

- Each object belongs to a unique class.
- Different objects may belong to different classes.
  - `System.out`
  - `"Hello world!"`
- An object that belongs to a class is called an *instance* of the class.
- A class may have more than one instance:
  - `"Hello world!"`
  - `"What's up Doc?"`
  - ...

# Classes (Continued)

- Each class has its own *Application Programming Interface* (API).
- The API describes how to use the class:
  - The names of the methods;
  - The types of the arguments;
  - The purpose of the arguments;
  - The return value;
  - Side effects;
  - ...
- The API defines a common protocol:

## Java

```
System.out.println( "Hello world!" );
System.err.println( "Fatal error." );
```

- Different classes may have different APIs.
  - E.g. an instance of the `String` class cannot print.

## Don't Try This at Home

```
"Hello world!".println( "What's up Doc?" );
```

# Variables

- Most programs require computations.
  - Add 13% VAT to the price;
  - Add 2 penalty points;
  - Determine the maximum input value;
  - ...
- A single computation may require many sub-computations.
- You (usually) store the results of a computation in a *variable*.
- A variable has several properties:
  - A name;
  - A memory location to store its value;
  - Its current value.
- To change a variable's value, you *assign* it a new value.

**Java**

```
⟨variable's name⟩ = ⟨expression that determines the value⟩;
```

# Variables

- Before you can use a variable, you must *declare* it.
- A variable declaration determines:
  - The variable's name;
  - The variable's type (the kind of its values);

**Java**

```
int counter;
double interest;
```

- A variable declaration may also determine the initial value;

**Java**

```
String greetings = "Hello world!";
```

# Assignment and Equality

- In mathematics you use = for equality.
- In Java you use = for assignment.

# Assignment and Equality

- In mathematics you use = for equality.
- In Java you use = for assignment.
- But assignment and equality are not the same.

# Assignment and Equality

- In mathematics you use = for equality.
- In Java you use = for assignment.
- But assignment and equality are not the same.
- The symbols are the “same” but they don’t mean the same.

# Assignment and Equality

- In mathematics you use = for equality.
- In Java you use = for assignment.
- But assignment and equality are not the same.
- The symbols are the "same" but they don't mean the same.
- Mathematical equality is commutative: if $a = b$, then $b = a$.

# Assignment and Equality

- In mathematics you use = for equality.
- In Java you use = for assignment.
- But assignment and equality are not the same.
- The symbols are the "same" but they don't mean the same.
- Mathematical equality is commutative: if $a = b$, then $b = a$.
- However, you can't write the following in Java:

## Don't Try This at Home

```
l = a; // ?
```

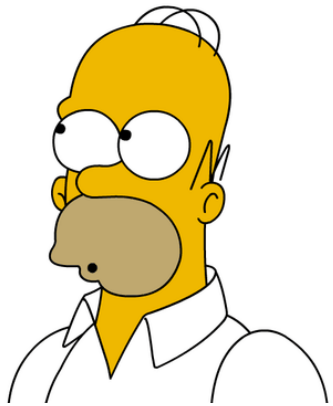
# Assignment and Equality

- In mathematics you use = for equality.
- In Java you use = for assignment.
- But assignment and equality are not the same.
- The symbols are the "same" but they don't mean the same.
- Mathematical equality is commutative: if $a = b$, then $b = a$.
- However, you can't write the following in Java:

## Don't Try This at Home

```
l = a; // ?
```

- In mathematics $a = a + 1$ is impossible.



# Assignment and Equality

- In mathematics you use = for equality.
- In Java you use = for assignment.
- But assignment and equality are not the same.
- The symbols are the "same" but they don't mean the same.
- Mathematical equality is commutative: if $a = b$, then $b = a$.
- However, you can't write the following in Java:

## Don't Try This at Home

```
1 = a; // ?
```

- In mathematics $a = a + 1$ is impossible.
- However, writing the following is valid in Java.

## Java

```
counter = counter + 1;
```

# Types

- A *type* is a collection of related values.
- E.g. `Java` has a whole range of numeric types.
  - whole numbers
    - `byte`;
    - `short`;
    - `int`;
    - `long`.
  - floating point
    - `float`;
    - `double`.
- If you want to assign a value to a variable,
  - The value must be in the the variable's type.
- This avoids logical errors:
  - `Dog dog = new Cat( "Felix" );?`
  - `Debit debit = new Credit( 666 );?`
- For whole numbers, the type `int` is usually a good.
- For floating point numbers, use `double`.

# Operations on Numbers

unary plus `+ ⟨operand⟩;`
unary minus `- ⟨operand⟩;`
adding `⟨operand #1⟩ + ⟨operand #2⟩;`
subtracting `⟨operand #1⟩ - ⟨operand #2⟩;`
multiplying `⟨operand #1⟩ * ⟨operand #2⟩;`
dividing `⟨operand #1⟩ / ⟨operand #2⟩;`
...

- Multiplicative operators bind more tightly:
  - `a * b + c` equals `(a * b) + c`.
  - `a / b + c` equals `(a / b) + c`.

# Primitive Types and Object Reference Types

- A type starting with a lowercase letter is a *primitive* type.
- E.g. `int`, `bool`, `char`, `float`, ...
- These values are bit patterns with well-defined operations.

**Java**

```
int lucky = 2 * 21;
```

- Types starting with an uppercase letter are *object/reference* types.
  - `Dog`, `Cat`, ...
  - Object reference values reference objects.

## Java

```
Dog lucky = new Terrier( );
lucky.bark( );
```

  - Primitive type values.don't reference obects:

## Don't Try This at Home

```
int lucky = 42;
luck.add( );
```

- `Java` also has *numeric* object reference classes.

## Java

```
Integer number = new Integer( 1 );
System.out.println( number.toString( ) );
```

  - Best view these types as wrapper classes for primitive type values.

# Wrapper Classes

- Java has a *wrapper class* for each primitive type.

  `Integer` For `int`s:
  - `final Integer iRef = new Integer( 42 );`
  - `final int val = iRef.intValue( );`

  `Double` For `double`s:
  - `final Double dRef = new Double( 3.14 );`
  - `final double val = dRef.doubleValue( );`

  `Boolean` For `boolean`s:
  - `final Boolean bRef = new Boolean( true );`
  - `final boolean val = bRef.booleanValue( );`

  ....

# Autoboxing and Unboxing

- Writing code to convert to and from wrapper classes is tedious.
  - `new Integer( 42 )`, `lucky.intValue( )`, ...
  - You must type more.
  - It increases the code size.
- That's why `Java` automates (some) conversions.
  - Automatic conversion to the wrapper class is called *autoboxing*.
  - Automatic conversion from the wrapper class is called *unboxing*.
- The conversion is done at runtime.

# Autoboxing

- Let `val` be an value with primitive type type.
  - If you use `val` and `Java` expects an object, `Java` will autobox `val`.
- The type of `val` determines the wrapper class:
  - `int` $\mapsto$ `Integer`;
  - `double` $\mapsto$ `Double`;
  - `boolean` $\mapsto$ `Boolean`;
  - ...

# Autoboxing

## Object Reference Expected in RHS

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Object Reference Provided in RHS

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Type of Object Reference in RHS is Correct

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Hunkey Dory

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Object Reference Expected in RHS

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Primitive Value Provided

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Autoboxing to the Rescue

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Primitive Type value is `int`

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## `int` Converted to its Reference Type: `Integer`

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Type of Object Reference in RHS is Correct

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Hunkey Dory

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Object Reference Expected in RHS

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Primitive Value Provided

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Autoboxing to the Rescue?

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Primitive Type value is `int`

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## `int` Converted to its Reference Type: `Integer`

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Wrong Type!

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Autoboxing

## Compile-Time Error!!

- `Integer lucky = new Integer( 42 );`
- `Integer lucky = 42; // autoboxing`
- `Double devil = 666; // doesn't work`

# Unboxing

- Unboxing turns wrapper class objects to primitive type values.
- The wrapper class type determines the primitive type.
  - `Integer` $\mapsto$ `int`;
  - `Double` $\mapsto$ `double`;
  - `Boolean` $\mapsto$ `boolean`;
  - ...
- The conversion is done at runtime.

**Java**

```
int multiplicand = lucky.intValue( );
int multiplier = lucky; // unboxing
int square = multiplicand * multiplier;
```

# Caching

- Java *caches* a limited number of wrapper class values.
- Guarantees shallow equality for small number of boxed values.
  - If `o1.equals( o2 )` then `o1 == o2`.
- For example, `new Integer( 0 ) == new Integer( 0 )`.
- In general this may not always work:
  - Almost always: `new Integer( 666 ) != new Integer( 666 )`.
- Caching is implemented because it saves memory.
- In general caching works for "small" primitive values.

| | |
|---|---|
| `boolean:` | `true` and `false`. |
| `byte:` | 0–255. |
| `char:` | `\u0000`–`\u007f`. |
| `short:` | -128, -127, ..., 127. |
| `int:` | -128, -127, ..., 127. |

# Constant Variables

- A *constant* (variable) can only be assigned a value once.
- You declare a constant by adding the keyword `final`.

**Java**

```
final int ANSWER = 42;
```

- Making a variable constant is a form of documentation.
- It lets the compiler help you detect logic errors:

**Java**

```
final double ACCELLERATION = 9.8;
...
ACCELLERATION = 9.9;
```

# Using Variables in Methods

- You cannot use an unassigned variable in a method.

## Don't Try This at Home

```
int number;
int square = number * number;
```

# Comments

- A *comment* is text that is ignored by the compiler.
- Comments have several purposes:
  - They describe the purpose of a variable or a method.
  - They describe a relationship between two or more variables.
    - This is called an invariant.
  - They are used to create API documentation.
- You should always document your programs.

# One Line Comments

## Java

```
// number of centimetres per inch
final double CENTIMETRES_PER_INCH = 2.56;
```

# Multi-Line Line Comments

## Java

```
/* Encrypted user password.
 * Use the changePassword( ) method to change the password.
 */
String password;
```

# JavaDoc Comments

## Java

```
/**
 * …
 */
```

# Variable Names

- Use names that are meaningful.
- The name should describe the variable's purpose.
- By convention each variable name should be a noun.

non-constant
- Each name should start with a lowercase letter.
- The rest should be letters and digits.
- At word boundaries, you use an uppercase letter.
- All other letters should be lowercase.
- E.g. `sum`, `currentColour`, ...

constant
- Use sequences of words, digits, and underscores.
- Each word is spelt with uppercase letters.
- At word boundaries, you use an underscore.
- E.g. `CENT`, `CENTIMETRES_PER_INCH`, ....

# Choosing Variable Names

- Variable names should be descriptive.
- This is a form of documentation:
  - It helps you remember what the variable does.
  - It helps others understand the purpose of the variable.
- Choosing a good name helps you understand the purpose.
- Always think about the purpose a variable should have.
- When you know the purpose, the name will follow.
  - A counter variable: `int counter;`
  - A bank account: `Account account;`
  - The wheel of a unicycle: `Wheel wheel;`
  - ...

# Choosing Variable Names

- Variable names should be descriptive.
- This is a form of documentation:
  - It helps you remember what the variable does.
  - It helps others understand the purpose of the variable.
- Choosing a good name helps you understand the purpose.
- Always think about the purpose a variable should have.
- When you know the purpose, the name will follow.
  - A counter variable: `int counter;`
  - A bank account: `Account account;`
  - The wheel of a unicycle: `Wheel wheel;`
  - ...
- If you can't find a proper name for a variable

# Choosing Variable Names

- Variable names should be descriptive.
- This is a form of documentation:
  - It helps you remember what the variable does.
  - It helps others understand the purpose of the variable.
- Choosing a good name helps you understand the purpose.
- Always think about the purpose a variable should have.
- When you know the purpose, the name will follow.
  - A counter variable: `int counter;`
  - A bank account: `Account account;`
  - The wheel of a unicycle: `Wheel wheel;`
  - ...
- If you can't find a proper name for a variable:
  - You don't really know its purpose.

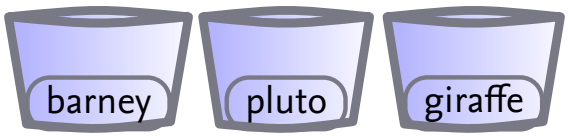
# Choosing Variable Names

- Variable names should be descriptive.
- This is a form of documentation:
  - It helps you remember what the variable does.
  - It helps others understand the purpose of the variable.
- Choosing a good name helps you understand the purpose.
- Always think about the purpose a variable should have.
- When you know the purpose, the name will follow.
  - A counter variable: `int counter;`
  - A bank account: `Account account;`
  - The wheel of a unicycle: `Wheel wheel;`
  - ...
- If you can't find a proper name for a variable:
  - You don't really know its purpose.
  - You may as well get rid of the variable.

# Java Cares about its Types

## Java

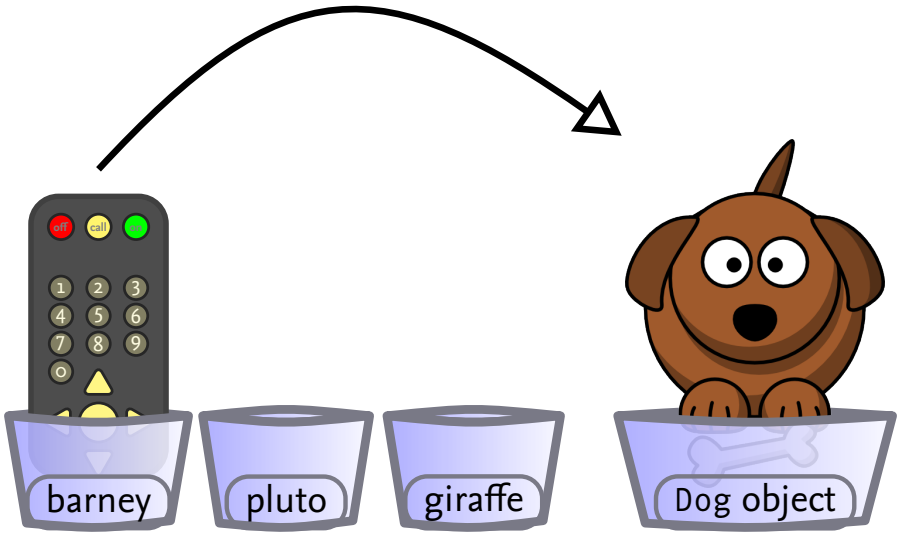
```
Dog barney = new Dog( );
Dog pluto = new Dog( );
Giraffe giraffe = new Giraffe( );
```



# Java Cares about its Types

## Java

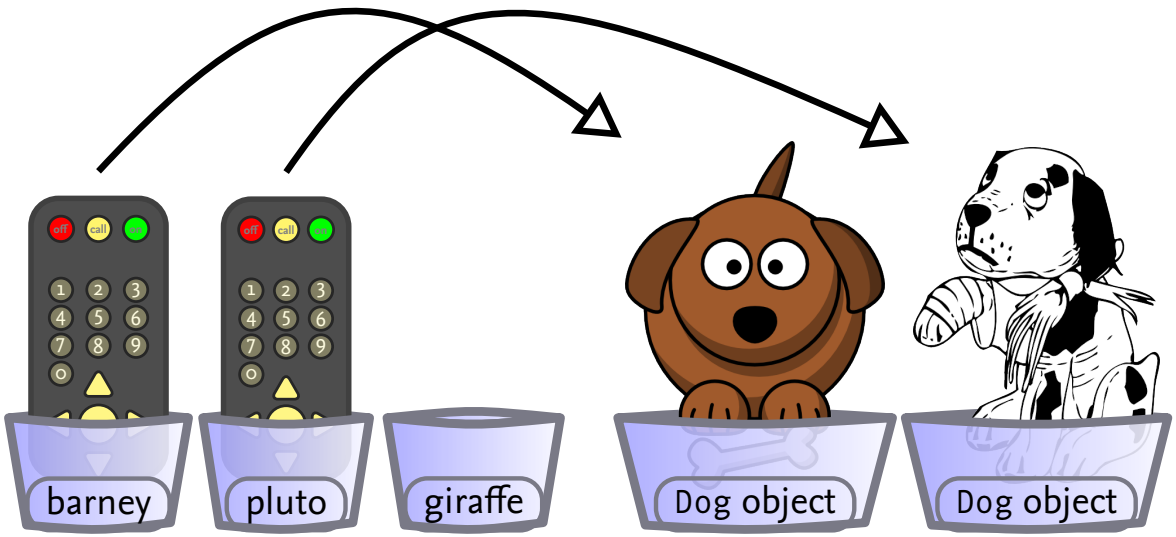
```
Dog barney = new Dog( );
Dog pluto = new Dog( );
Giraffe giraffe = new Giraffe( );
```



# Java Cares about its Types

## Java

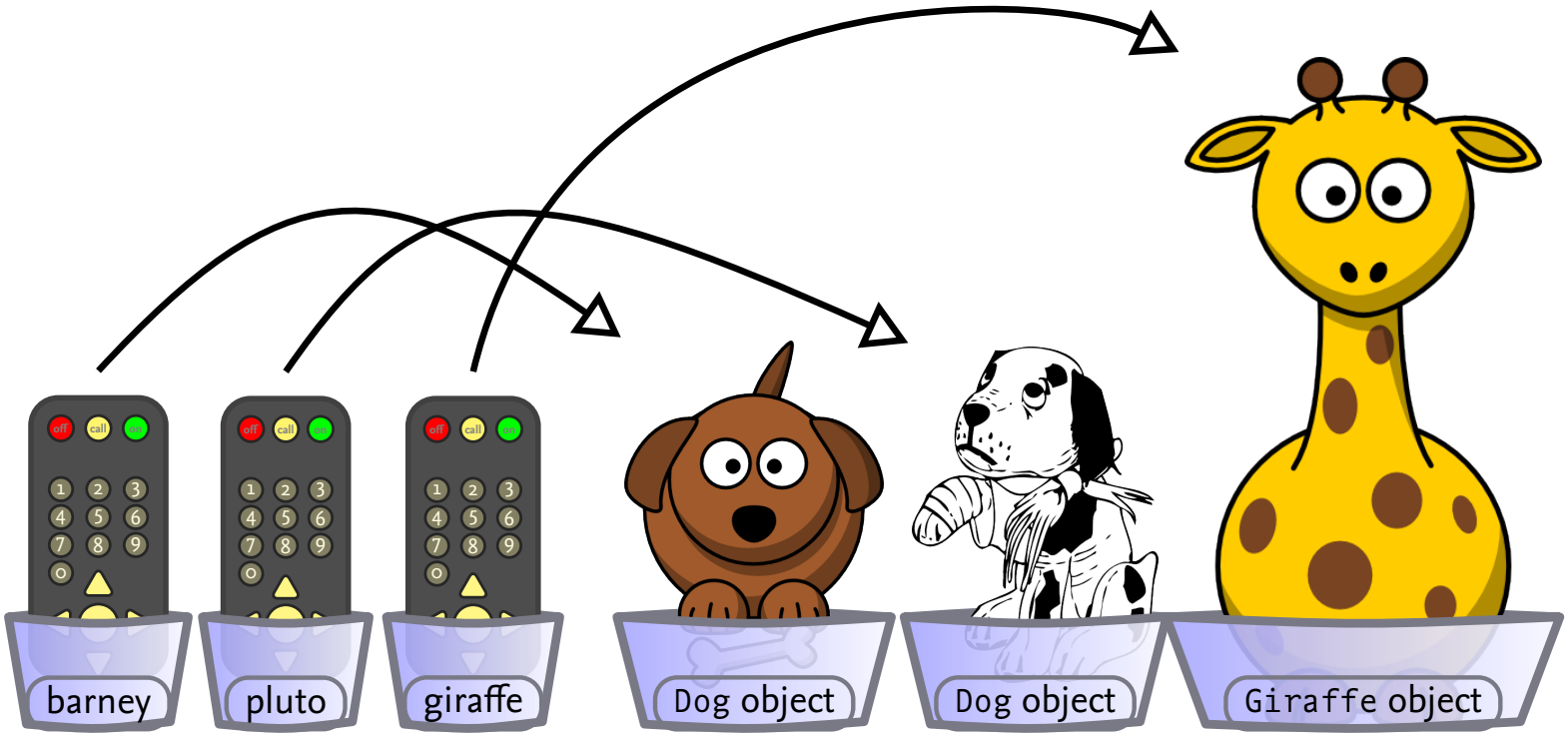
```
Dog barney = new Dog( );
Dog pluto = new Dog( );
Giraffe giraffe = new Giraffe( );
```



# Java Cares about its Types

## Java

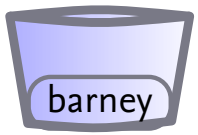
```
Dog barney = new Dog( );
Dog pluto = new Dog( );
Giraffe giraffe = new Giraffe( );
```



# Types Really Matter

## Don't Try This at Home

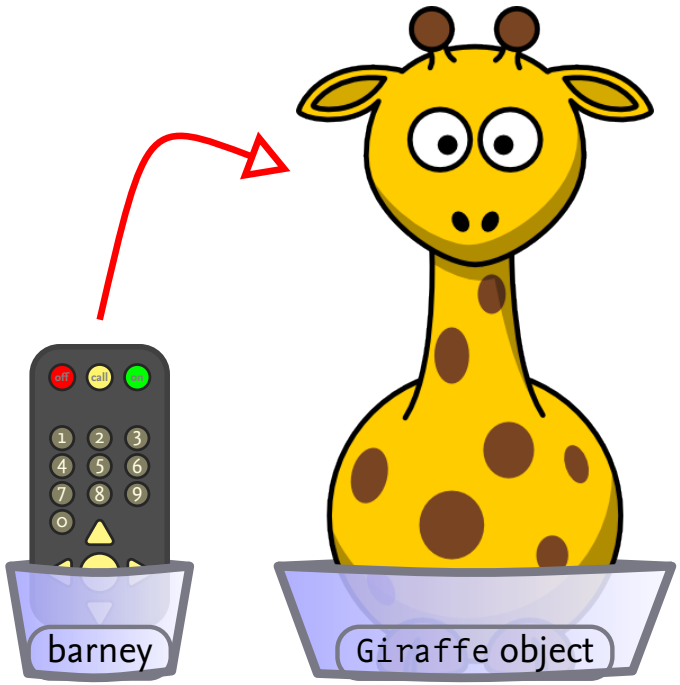
```
Dog barney = new Giraffe( );
```



# Types Really Matter

## Don't Try This at Home

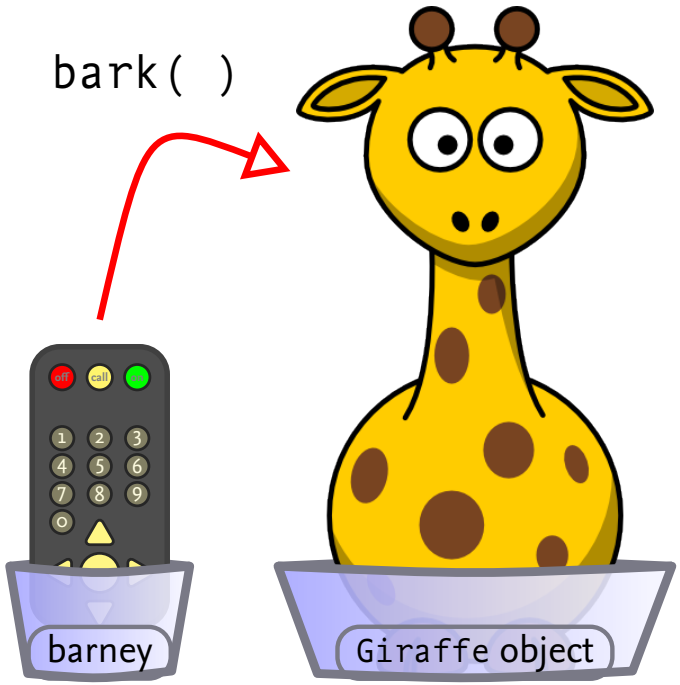
```
Dog barney = new Giraffe( );
```



# Types Really Matter

## Don't Try This at Home

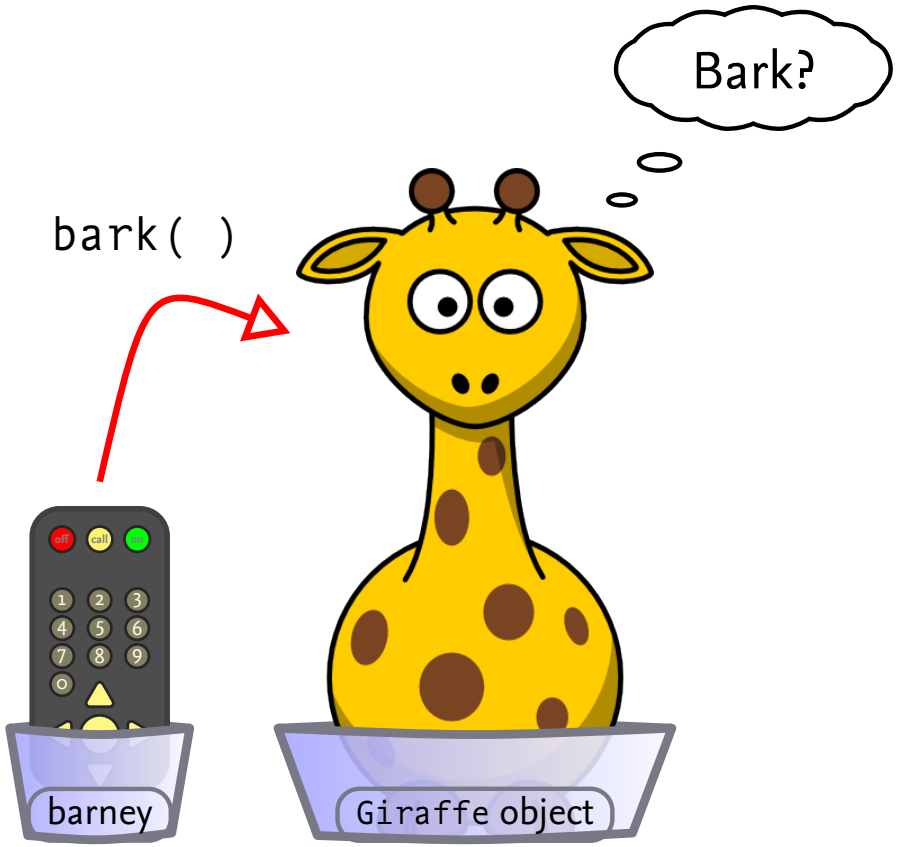
```
Dog barney = new Giraffe( );
barney.bark( );
```



# Types Really Matter

## Don't Try This at Home

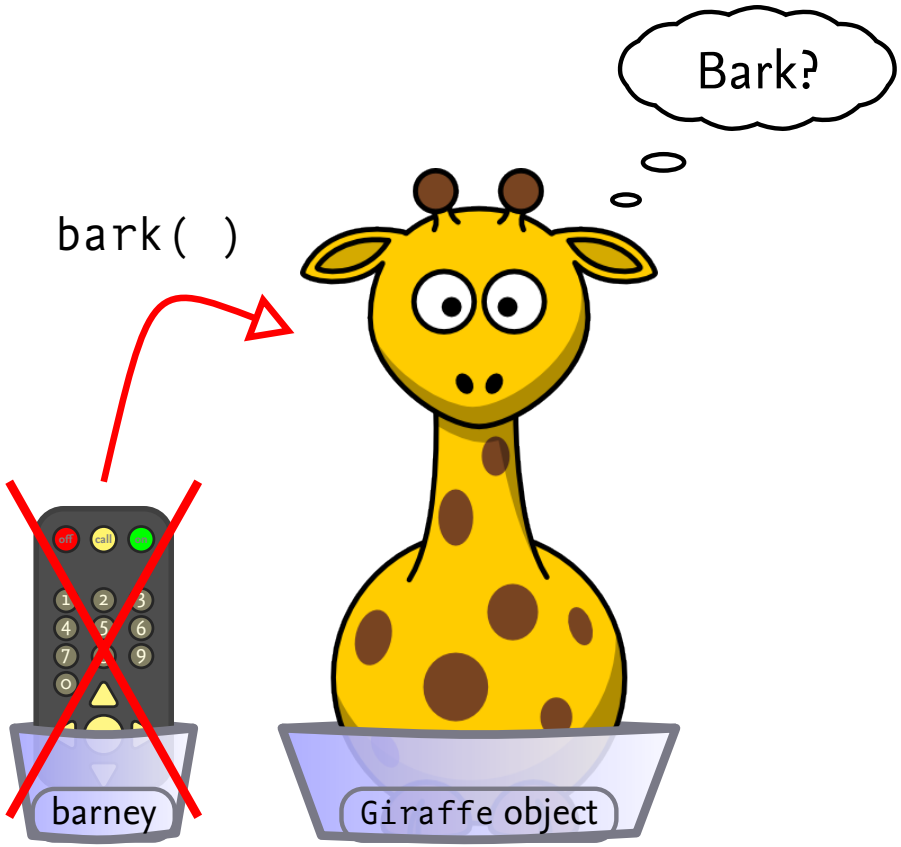
```
Dog barney = new Giraffe( );
barney.bark( ); // ???
```



# Types Really Matter

## Don't Try This at Home

```
Dog barney = new Giraffe( ); // Impossible
barney.bark( ); // ???
```



# Working with Objects

- Before you can use an object, you must construct (create) it.
- To construct an object, you call its constructor.
  - The constructor constructs and initialises the object.
- There may be different ways to construct an object.

**Java**

```
final Rectangle bar = new Rectangle( x, y, width, height );
```

# Constructing the Rectangle

## Java

```
Rectangle bar = new Rectangle( x, y, width, height )
```

1. The new operator creates memory to represents the object;
2. The constructor uses its arguments to initialise the object;
3. The constructor returns a *reference* to the object;
4. The reference is assigned to the object reference value bar.
5. The reference may be used to call the object's instance methods.

# Method Declarations

- To define/declare a method you provide:
  - The name of the method;
  - The return type;
  - The names and types of the formal parameters;
  - The types of the formal parameters.

Java

```
public int getWidth( ) { /* Implementation omitted. */ }
```

- You use `void` for a method without return value.

Java

```
public void println( String output ) { /* Implementation omitted. */ }
```

- If the argument types are different, the names may overlap.
  - This is called *overloading:*

Java

```
public void println( int output ) { /* Implementation omitted. */ }
```

# Accessor and Mutator Methods

- A method that returns information about an object without modifying the object is an *accessor method.*
  - `double width = rectangle.getWidth( );`
- A method that modifies an object's instance variables is a *mutator method.*
  - `rectangle.setWidth( 4.0 );`

# Implementing a Tally Counter Class

- Let's implement a tally counter object class.
- The name of the class should be a noun.
  - The name should start with an uppercase letter.
  - The name should continue with letters and digits.
  - At each word boundary, you use an uppercase letter.
  - All other letters should be lowercase.
  - The name should describe the instances of the class.
  - For example, `StringBuilder`, `FullAdder`, ...

# State and Behaviour

- Let's use `Counter` for our class name.
- How do we implement the class?
- We must determine what the `Counter` instances do and know.
- What the instance does is its *behaviour*.
  - Object behaviour is implemented as *instance methods*.
- What the instance knows is its *state*.
  - Object state is implemented as *instance variables*.

# State and Behaviour

- Let's use `Counter` for our class name.
- How do we implement the class?
- We must determine what the `Counter` instances do and know.
- What the instance does is its *behaviour*.
  - Object behaviour is implemented as *instance methods*.
- What the instance knows is its *state*.
  - Object state is implemented as *instance variables*.
- Too much (object) state slows down the JVM.

# State and Behaviour

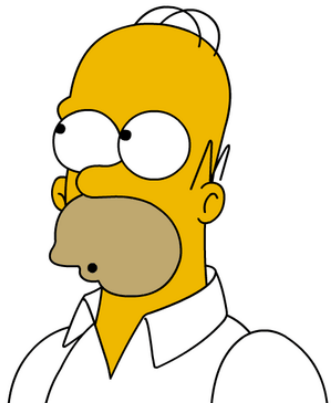
- Let's use `Counter` for our class name.
- How do we implement the class?
- We must determine what the `Counter` instances do and know.
- What the instance does is its *behaviour*.
  - Object behaviour is implemented as *instance methods*.
- What the instance knows is its *state*.
  - Object state is implemented as *instance variables*.
- Too much (object) state slows down the JVM.
- An object's behaviour should determine its state:

# State and Behaviour

- Let's use `Counter` for our class name.
- How do we implement the class?
- We must determine what the `Counter` instances do and know.
- What the instance does is its *behaviour*.
  - Object behaviour is implemented as *instance methods*.
- What the instance knows is its *state*.
  - Object state is implemented as *instance variables*.
- Too much (object) state slows down the JVM.
- An object's behaviour should determine its state:
  - Never, *ever* start with object state.

# State and Behaviour

- Let's use `Counter` for our class name.
- How do we implement the class?
- We must determine what the `Counter` instances do and know.
- What the instance does is its *behaviour*.
  - Object behaviour is implemented as *instance methods*.
- What the instance knows is its *state*.
  - Object state is implemented as *instance variables*.
- Too much (object) state slows down the JVM.
- An object's behaviour should determine its state:
  - Never, *ever* start with object state.
  - Start thinking about the behaviour.



# State and Behaviour

- Let's use `Counter` for our class name.
- How do we implement the class?
- We must determine what the `Counter` instances do and know.
- What the instance does is its *behaviour*.
  - Object behaviour is implemented as *instance methods*.
- What the instance knows is its *state*.
  - Object state is implemented as *instance variables*.
- Too much (object) state slows down the JVM.
- An object's behaviour should determine its state:
  - Never, *ever* start with object state.
  - Start thinking about the behaviour.
  - If behaviour requires state, you implement the state.

# State and Behaviour

- Let's use `Counter` for our class name.
- How do we implement the class?
- We must determine what the `Counter` instances do and know.
- What the instance does is its *behaviour*.
  - Object behaviour is implemented as *instance methods*.
- What the instance knows is its *state*.
  - Object state is implemented as *instance variables*.
- Too much (object) state slows down the JVM.
- An object's behaviour should determine its state:
  - Never, *ever* start with object state.
  - Start thinking about the behaviour.
  - If behaviour requires state, you implement the st
  - Otherwise, you don't.

# Behaviour

## What Should a `Counter` Object Do?

# Behaviour

## What Should a `Counter` Object Do?

- Compute its next counter value:
  - `public void incrementValue( )`
- Return its current counter value:
  - `public int getValue( )`

# Behaviour

## What Should a `Counter` Object Do?

- Compute its next counter value:
  - `public void incrementValue( )`
- Return its current counter value:
  - `public int getValue( )`

# Behaviour

## What Should a `Counter` Object Do? This indicates its state.

- Compute its next counter value:
  - `public void incrementValue( )`
- Return its current counter value:
  - `public int getValue( )`

# What Should a `Counter` Object Know?

# What Should a `Counter` Object Know?

- Its counter value:
  - `private int value;`

# The Class

## Class Definition

### Java

```
// Class for representing tally counter objects.
public class Counter {
    // The current tally counter value.
    private int value;

    // Returns the current counter value.
    public int getValue( ) {
        return value;
    }

    // Increment the counter value.
    public void incrementValue( ) {
        value = value + 1;
    }
}
```

# The Class

## Instance Attribute Declaration

### Java

```
// Class for representing tally counter objects.
public class Counter {
    // The current tally counter value.
    private int value;

    // Returns the current counter value.
    public int getValue( ) {
        return value;
    }

    // Increment the counter value.
    public void incrementValue( ) {
        value = value + 1;
    }
}
```

# The Class

## Instance Method Declarations

### Java

```
// Class for representing tally counter objects.
public class Counter {
    // The current tally counter value.
    private int value;

    // Returns the current counter value.
    public int getValue( ) {
        return value;
    }

    // Increment the counter value.
    public void incrementValue( ) {
        value = value + 1;
    }
}
```

# The Class

## Access/Visibility Specifiers

### Java

```
// Class for representing tally counter objects.
public class Counter {
    // The current tally counter value.
    private int value;

    // Returns the current counter value.
    public int getValue( ) {
        return value;
    }

    // Increment the counter value.
    public void incrementValue( ) {
        value = value + 1;
    }
}
```

# The Class

## Types

### Java

```
// Class for representing tally counter objects.
public class Counter {
    // The current tally counter value.
    private int value;

    // Returns the current counter value.
    public int getValue( ) {
        return value;
    }

    // Increment the counter value.
    public void incrementValue( ) {
        value = value + 1;
    }
}
```

# The Class

## Comments: You should use JavaDoc Here

### Java

```
// Class for representing tally counter objects.
public class Counter {
    // The current tally counter value.
    private int value;

    // Returns the current counter value.
    public int getValue( ) {
        return value;
    }

    // Increment the counter value.
    public void incrementValue( ) {
        value = value + 1;
    }
}
```

# Instance Variables

- Each `Counter` object has its own `value` variable.
- Let's assume `tally` is a `Counter` object reference (variable):
  - To access its `value` you write `tally.value`.
- The `Counter` object owns the variable.
- Different `Counter` objects may have different values for `value`.

# Instance Methods

- `Counter` objects can call `Counter` instance methods.
- Calling them is similar to accessing the instance variable:
  - `tally.incrementValue( );`
  - `int current = tally.getValue( );`

# Encapsulation

- Objects should be self-governing.
- They should control their own instance variables.
- An object is self-governing if its instance variables are `private`.
- This is called *hiding* the instance variables.
  - Variable hiding prevents direct variable access by external clients.
- Hiding the instance variables makes the object self-contained.
  - It's as if the object's instance variables are in a capsule.
  - This is why instance variable hiding is usually called *encapsulation*.

# Why Do We Need Encapulation?

- Direct attribute access is unsafe/dangerous.
  - A malicious external agent may corrupt the attribute's value.
- Encapsulation simplifies the complexity of the API.
  - Makes learning the API easier.
  - Makes using the API easier.
  - Makes designing the API easier.
  - Makes reasoning about the API easier.
  - Makes testing the API easier.
  - Makes maintaining the API easier.

# Contract

- We hide all instance variables.
- We hide all methods that aren't/shouldn't be part of the API.

# Hiding Methods

- Java also lets you hide method declarations.

**Java**

```
public int squareOfAnswer( ) {
    return answer( ) * answer( );
}

private int answer( ) {
    return 42;
}
```

- Hiding methods has similar advantages as hiding attributes.

# Automatic Variables

- A variable that is declared in a method is called *automatic.*
  - It only lives for the lifespan of its block during its method call.

# Automatic Variables

- A variable that is declared in a method is called *automatic.*
  - It only lives for the lifespan of its block during its method call.
- Use automatic variables for intermediate computations.

# Automatic Variables

- A variable that is declared in a method is called *automatic.*
  - It only lives for the lifespan of its block during its method call.
- Use automatic variables for intermediate computations.
- Don't use instance attributes for intermediate computations.

# Introduction

- Arrays are a special data type in `Java`.
- Arrays are objects that contain other things.
- There are two kinds of arrays:
  1. Arrays consisting of primitive type values;
  2. Arrays consisting of object reference values;
- The type of the array determines the type of its values.
- Before you can use an array you must create it (it's an object).
  - When doing this, you must specify the array's length.
  - The length remains fixed.
- You can put things into the array.
- You can retrieve things from the array.
- You can only access arrays with index values:
  - Only `int` index values are allowed.
  - They must be non-negative;
  - They must be smaller than the length of the array.

# Initialisation

## Java

```
final int[] numbers = new int[ 10 ];
System.out.println( "length of numbers: " + numbers.length );

final String[] words = new String[ 5 ];
System.out.println( "length of words: " + words.length );
```

# Initialisation

## Java

```
final int[] numbers = new int[ 10 ];
System.out.println( "length of numbers: " + numbers.length );

final String[] words = new String[ 5 ];
System.out.println( "length of words: " + words.length );
```

# Getting Stuff from the Array

- An array is best viewed as a tray/sequence with cups.
- Each cup has a number: 0, 1, ...
- The cups contain what's in the array:
  - `Object` references.
- The number of cups is the length of the array.
- Let `array` be a `Java` array.
- Then `array[ i ]` is the `i`th cup of `array`.

## Java

```
final int[] numbers = new int[ 10 ];
...
System.out.println( "The first value is " + numbers[ 0 ] );
System.out.println( "The last value is " + numbers[ 9 ] );
```

# Getting Stuff from the Array

- An array is best viewed as a tray/sequence with cups.
- Each cup has a number: 0, 1, ...
- The cups contain what's in the array:
  - `Object` references.
- The number of cups is the length of the array.
- Let `array` be a `Java` array.
- Then `array[ i ]` is the `i`th cup of `array`.

**Java**

```
final int[] numbers = new int[ 10 ];
...
System.out.println( "The first value is " + numbers[ 0 ] );
System.out.println( "The last value is " + numbers[ 9 ] );
```

# Getting Stuff from the Array

- An array is best viewed as a tray/sequence with cups.
- Each cup has a number: 0, 1, ...
- The cups contain what's in the array:
  - `Object` references.
- The number of cups is the length of the array.
- Let `array` be a `Java` array.
- Then `array[ i ]` is the `i`th cup of `array`.

**Java**

```
final int[] numbers = new int[ 10 ];
...
System.out.println( "The first value is " + numbers[ 0 ] );
System.out.println( "The last value is " + numbers[ 9 ] );
```

# Writing Stuff to the Array

- The notation `array[ index ]` works just as with getting.
- Cups in the arrays work just like variables, so
  - `array[ index ] = value` assigns a `value` to the "`index`th" cup.

# Writing Stuff to the Array

- The notation `array[ index ]` works just as with getting.
- Cups in the arrays work just like variables, so
  - `array[ index ] = value` assigns a `value` to the "`index`th" cup.

## Java

```
final int[] numbers = new int[ 10 ];

numbers[ 0 ] = 1;
numbers[ 9 ] = 42;
System.out.println( numbers[ 0 ] + " == 1" );
System.out.println( numbers[ 9 ] + " == 42" );
```

# Writing Stuff to the Array

- The notation `array[ index ]` works just as with getting.
- Cups in the arrays work just like variables, so
  - `array[ index ] = value` assigns a `value` to the "`index`th" cup.

## Java

```
final int[] numbers = new int[ 10 ];

numbers[ 0 ] = 1;
numbers[ 9 ] = 42;
System.out.println( numbers[ 0 ] + " == 1" );
System.out.println( numbers[ 9 ] + " == 42" );
```

# Writing Stuff to the Array

- The notation `array[ index ]` works just as with getting.
- Cups in the arrays work just like variables, so
  - `array[ index ] = value` assigns a `value` to the "`index`th" cup.

## Java

```
final int[] numbers = new int[ 10 ];

numbers[ 0 ] = 1;
numbers[ 9 ] = 42;
System.out.println( numbers[ 0 ] + " == 1" );
System.out.println( numbers[ 9 ] + " == 42" );
```

# Writing Stuff to the Array

- The notation `array[ index ]` works just as with getting.
- Cups in the arrays work just like variables, so
  - `array[ index ] = value` assigns a `value` to the "`index`th" cup.

## Java

```
final int[] numbers = new int[ 10 ];

numbers[ 0 ] = 1;
numbers[ 9 ] = 42;
System.out.println( numbers[ 0 ] + " == 1" );
System.out.println( numbers[ 9 ] + " == 42" );
```

# Default Values

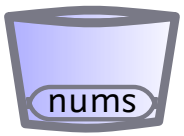
- When the JVM creates an array, it initialises the array's contents.
- Each cup in the array is filled with the same value.
- This value depends on the type of the array.

```
Numeric 0;
boolean false;
   char '\u0000';
 Object null.
```

# Arrays with Primitive Type Values

## Java

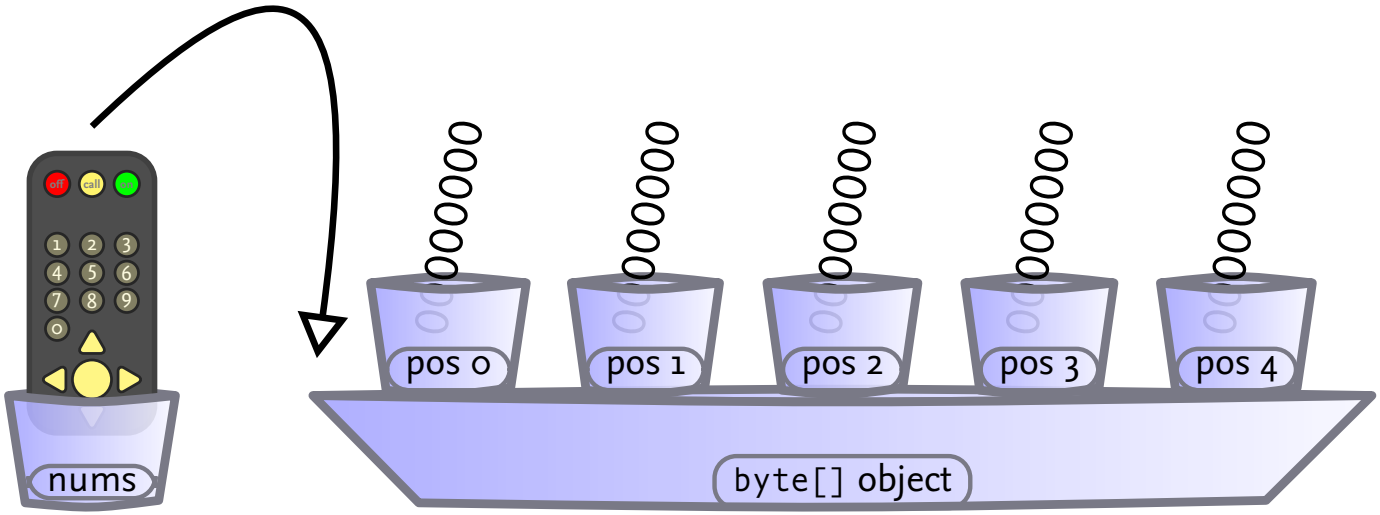
```
byte[] nums = new byte[ 5 ];
nums[ 1 ] = 4;
nums[ 4 ] = 17;
```



# Arrays with Primitive Type Values

## Java

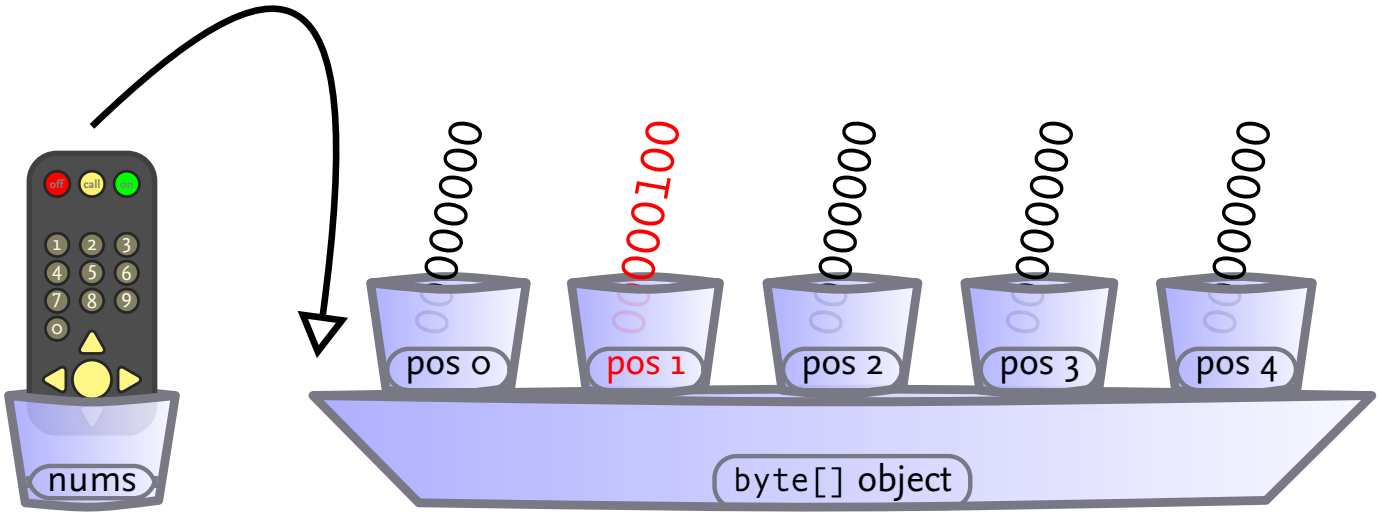
```
byte[] nums = new byte[ 5 ];
nums[ 1 ] = 4;
nums[ 4 ] = 17;
```



# Arrays with Primitive Type Values

## Java

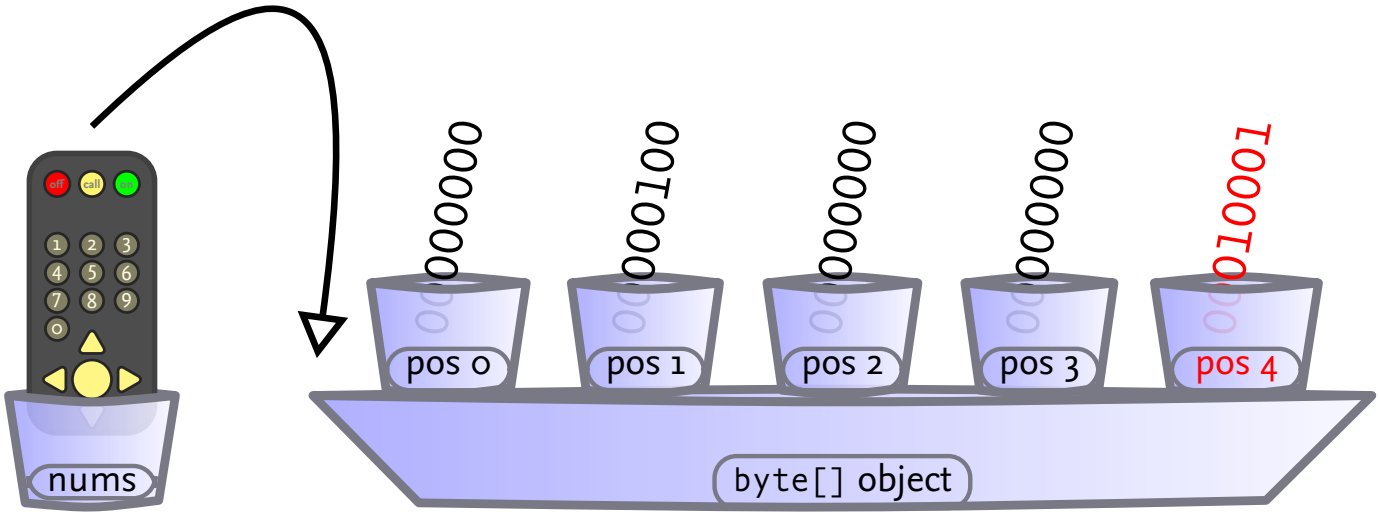
```
byte[] nums = new byte[ 5 ];
nums[ 1 ] = 4;
nums[ 4 ] = 17;
```



# Arrays with Primitive Type Values

## Java

```
byte[] nums = new byte[ 5 ];
nums[ 1 ] = 4;
nums[ 4 ] = 17;
```



# Arrays with Objects

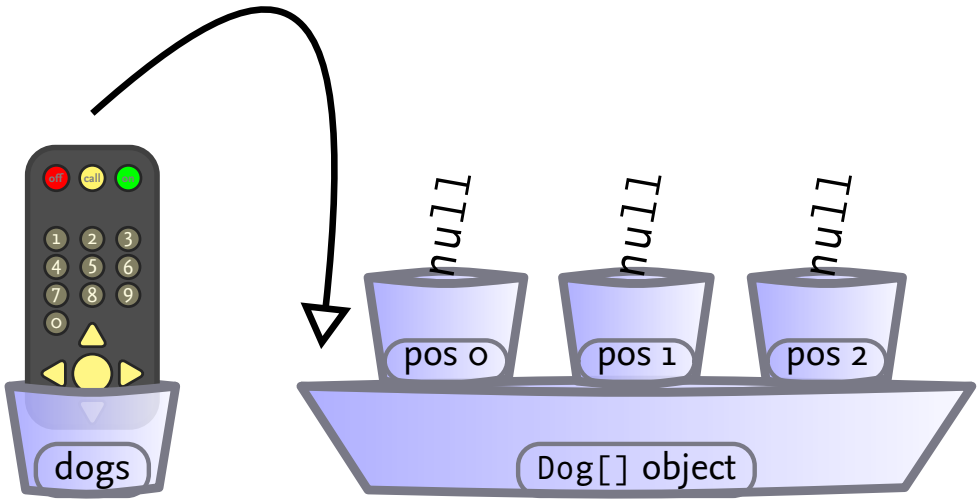
## Java

```
Dog[] dogs = new Dog[ 3 ];
dogs[ 1 ] = new Dog( );
dogs[ 1 ].bark( );
dogs[ 0 ].bark( );
```

# Arrays with Objects

## Java

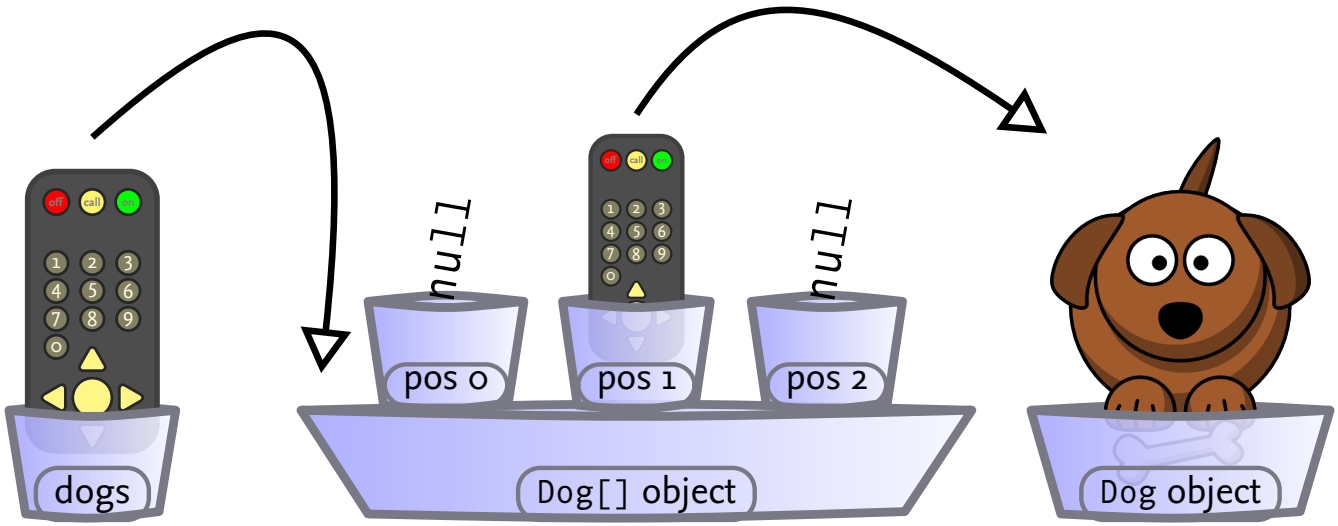
```
Dog[] dogs = new Dog[ 3 ];
dogs[ 1 ] = new Dog( );
dogs[ 1 ].bark( );
dogs[ 0 ].bark( );
```



# Arrays with Objects

## Java

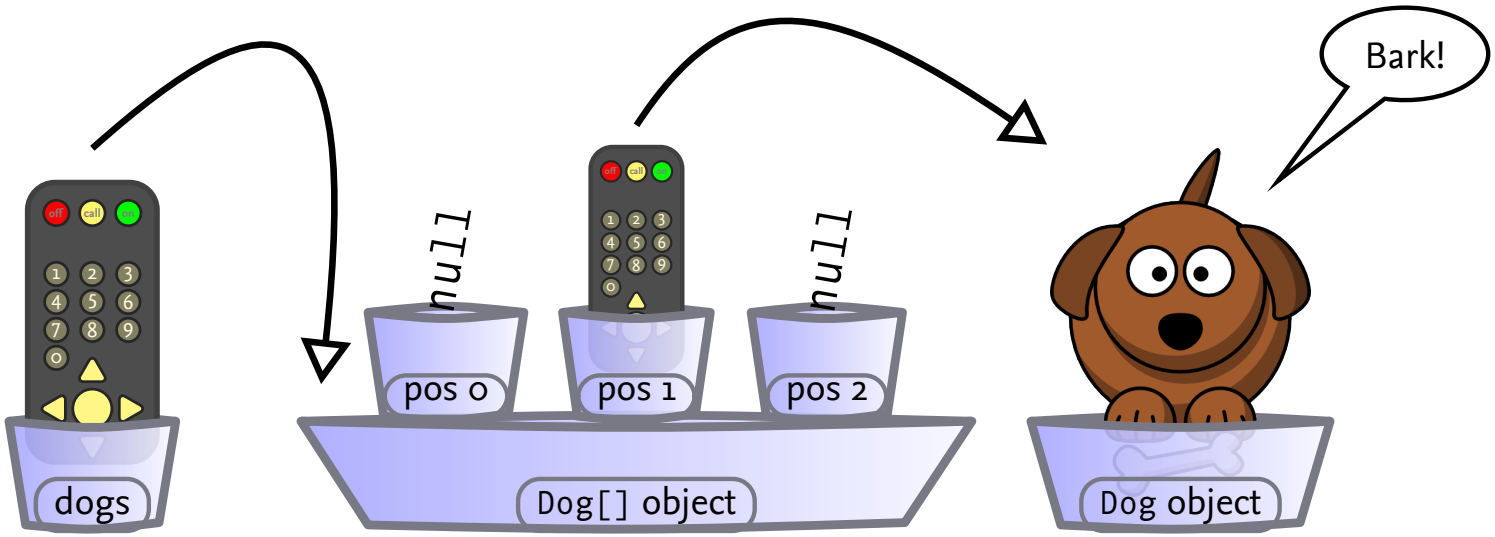
```
Dog[] dogs = new Dog[ 3 ];
dogs[ 1 ] = new Dog( );
dogs[ 1 ].bark( );
dogs[ 0 ].bark( );
```



# Arrays with Objects

## Java

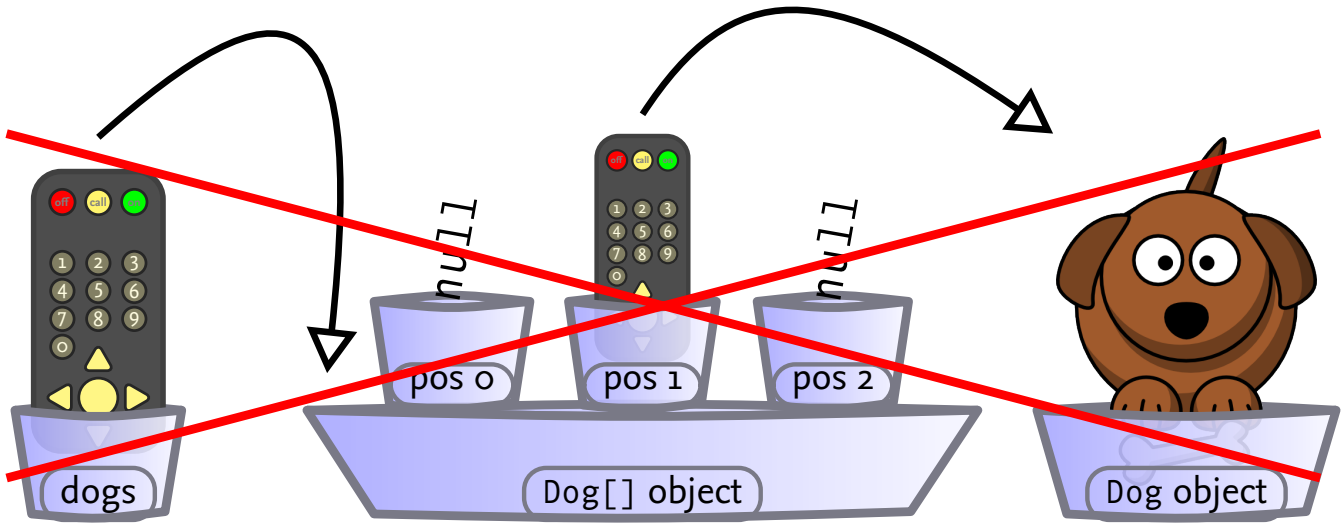
```
Dog[] dogs = new Dog[ 3 ];
dogs[ 1 ] = new Dog( );
dogs[ 1 ].bark( );
dogs[ 0 ].bark( );
```



# Arrays with Objects

## Java

```
Dog[] dogs = new Dog[ 3 ];
dogs[ 1 ] = new Dog( );
dogs[ 1 ].bark( );
dogs[ 0 ].bark( ); // Run-time error!
```



# Arrays do Not Grow

- The `length` attribute of a `Java` array is `final`.
- So you cannot assign values to ⟨`array`⟩`.length`.
- The minimum size of any array is `0`.
- The maximum size of any array is `Integer.MAX_VALUE`.

# Partially Filled Arrays

- You must fill the array before you can use it.
- You usually start filling at the bottom (index 0).
- Then fill the next position (index 1).
- And so on.
- You need a counter to keep track of the current index.

## Java

```
final Scanner scanner = new Scanner( System.in );
final int[] values = new int[ scanner.nextInt( ) ];

int size = 0;
int next = 0;
while ((size != values.length) && (next >= 0)) {
    System.err.println( "Next value (negative value to stop): " );
    next = scanner.next( );
    if (next >= 0) {
        values[ size++ ] = next;
    }
}

final double percentage = 100.0 * size / values.length );
System.out.println( "Percentage filled is " + percentage );
```

# Partially Filled Arrays

- You must fill the array before you can use it.
- You usually start filling at the bottom (index 0).
- Then fill the next position (index 1).
- And so on.
- You need a counter to keep track of the current index.

## Java

```
final Scanner scanner = new Scanner( System.in );
final int[] values = new int[ scanner.nextInt( ) ];

int size = 0;
int next = 0; // We need this to enter the loop.
while ((size != values.length) && (next >= 0)) {
    System.err.println( "Next value (negative value to stop): " );
    next = scanner.next( );
    if (next >= 0) {
        values[ size++ ] = next;
    }
}

final double percentage = 100.0 * size / values.length );
System.out.println( "Percentage filled is " + percentage );
```

# Common Errors

## Index too Large

### Don't Try This at Home

```
int[] values = new int[ 10 ];

values[ 10 ] = 1;
```

# Common Errors

## Index too Small

### Don't Try This at Home

```
int[] values = new int[ 10 ];

values[ -1 ] = 1;
```

# Common Errors

## Uninitialised Values

### Don't Try This at Home

```
String[] words = new String[ 10 ];

if (words[ 0 ].equals( "yes" )) {
    System.out.println( "This isn't printed." );
} else {
    System.out.println( "This also isn't printed." );
}
```

# Representing Bank Accounts

- Consider a bank account application.
- Each account has an owner and a balance.
  - We could represent the owners using a `String` array;
  - We could represent the balance using a `double` array.

# Parallel Array Implementation

## The `for` Loop Declares Its Own `index` Variable

### Java

```
public class AccountManager {
    private final String[] owners;
    private final double[] balances;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        owners = new String[ size ];
        balances = new double[ size ];
        for (int index = 0; index != size; index++) {
            owners[ index ] = scanner.next( );
            balances[ index ] = scanner.nextDouble( );
        }
    }

    ...
}
```

# Parallel Array Implementation

## The `for` Loop Declares Its Own `index` Variable

### Java

```
public class AccountManager {
    private final String[] owners;
    private final double[] balances;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        owners = new String[ size ];
        balances = new double[ size ];
        for (int index = 0; index != size; index++) {
            owners[ index ] = scanner.next( );
            balances[ index ] = scanner.nextDouble( );
        }
    }

    ...
}
```

# Parallel Array Implementation

## The `for` Loop Declares Its Own `index` Variable

### Java

```
public class AccountManager {
    private final String[] owners;
    private final double[] balances;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        this.owners = new String[ size ];
        this.balances = new double[ size ];
        for (int index = 0; index != size; index++) {
            owners[ index ] = scanner.next( );
            balances[ index ] = scanner.nextDouble( );
        }
    }

    ...
}
```

# Parallel Array Implementation

## The `for` Loop Declares Its Own `index` Variable

### Java

```
public class AccountManager {
    private final String[] owners;
    private final double[] balances;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        owners = new String[ size ];
        balances = new double[ size ];
        for (int index = 0; index != size; index++) {
            owners[ index ] = scanner.next( );
            balances[ index ] = scanner.nextDouble( );
        }
    }

    ...
}
```

# Class-Based Implementation

## Java

```
public class AccountManager {
    private final Account[] accounts;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        accounts = new Account[ size ];
        for (int index = 0; index != size; index++) {
            final String owner = scanner.next( );
            final double balance = scanner.nextDouble( );
            accounts[ index ] = new Account( owner, balance );
        }
    }

    ...
}
```

# Class-Based Implementation

## Java

```
public class AccountManager {
    private final Account[] accounts;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        accounts = new Account[ size ];
        for (int index = 0; index != size; index++) {
            final String owner = scanner.next( );
            final double balance = scanner.nextDouble( );
            accounts[ index ] = new Account( owner, balance );
        }
    }

    ...
}
```

# Class-Based Implementation

## Java

```
public class AccountManager {
    private final Account[] accounts;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        this.accounts = new Account[ size ];
        for (int index = 0; index != size; index++) {
            final String owner = scanner.next( );
            final double balance = scanner.nextDouble( );
            accounts[ index ] = new Account( owner, balance );
        }
    }

    ...
}
```

# Class-Based Implementation

## Java

```
public class AccountManager {
    private final Account[] accounts;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        accounts = new Account[ size ];
        for (int index = 0; index != size; index++) {
            final String owner = scanner.next( );
            final double balance = scanner.nextDouble( );
            accounts[ index ] = new Account( owner, balance );
        }
    }

    ...
}
```

# Comparison

**Stability** The parallel array implementation is "unstable:"

- Adding/removing attributes affects API of all methods that depend on them.

**Security** The parallel array implementation is not safe:

- Parallel array clients need access to all arrays:
  - `withdraw( owners, balances, nr, amount );`
  - This gives the client access to all account details.
  - They can even modify the array.
  - It violates encapsulation.
- Direct access for `Account` clients:
  - `account.withdraw( amount ).`
- Perhaps better to add service at `AccountManager` level:

**Java**

```
public void withdraw( final Account account, final double amount ) {
    if (⟨conditions are right⟩) {
        account.withdraw( amount );
    }
}
```

# Comparison

**Stability** The parallel array implementation is "unstable:"

- Adding/removing attributes affects API of all methods that depend on them.

**Security** The parallel array implementation is not safe:

- Parallel array clients need access to all arrays:
  - `withdraw( owners, balances, nr, amount );`
  - This gives the client access to all account details.
  - They can even modify the array.
  - It violates encapsulation.
- Direct access for `Account` clients:
  - `account.withdraw( amount ).`
- Perhaps better to add service at `AccountManager` level:

**Java**

```
public void withdraw( final Account account, final double amount ) {
    if (⟨conditions are right⟩) {
        account.withdraw( amount );
    }
}
```

# Comparison

**Stability** The parallel array implementation is "unstable:"

- Adding/removing attributes affects API of all methods that depend on them.

**Security** The parallel array implementation is not safe:

- Parallel array clients need access to all arrays:
  - `withdraw( owners, balances, nr, amount );`
  - This gives the client access to all account details.
  - They can even modify the array.
  - It violates encapsulation.
- Direct access for `Account` clients:
  - `account.withdraw( amount ).`
- Perhaps better to add service at `AccountManager` level:

**Java**

```
public void withdraw( final Account account, final double amount ) {
    if (⟨conditions are right⟩) {
        account.withdraw( amount );
    }
}
```

# Comparison

**Stability** The parallel array implementation is "unstable:"

- Adding/removing attributes affects API of all methods that depend on them.

**Security** The parallel array implementation is not safe:

- Parallel array clients need access to all arrays:
  - `withdraw( owners, balances, nr, amount );`
  - This gives the client access to all account details.
  - They can even modify the array.
  - It violates encapsulation.
- Direct access for `Account` clients:
  - `account.withdraw( amount ).`
- Perhaps better to add service at `AccountManager` level:

**Java**

```
public void withdraw( final Account account, final double amount ) {
    if (⟨conditions are right⟩) {
        account.withdraw( amount );
    }
}
```

# Comparison

**Stability** The parallel array implementation is "unstable:"

- Adding/removing attributes affects API of all methods that depend on them.

**Security** The parallel array implementation is not safe:

- Parallel array clients need access to all arrays:
  - `withdraw( owners, balances, nr, amount );`
  - This gives the client access to all account details.
  - They can even modify the array.
  - It violates encapsulation.
- Direct access for `Account` clients:
  - `account.withdraw( amount ).`
- Perhaps better to add service at `AccountManager` level:

**Java**

```
public void withdraw( final Account account, final double amount ) {
    if (⟨conditions are right⟩) {
        account.withdraw( amount );
    }
}
```

# Question Time

Questions Anybody?

# For Next Monday

- Study the presentation.

# Acknowledgements


- This lecture is partially based on
  - [Sierra, and Bates 2004, Chapters 2 and 3].
  - [Horstmann 2013].

# Bibliography I



Horstmann, Cay S. [2013]. *Big Java, Early Objects*. International Student Version. Wiley. ISBN: 978-1-118-31877-5.


Sierra, Kathy, and Bert Bates [2004]. *Head First Java*. O'Reilly. ISBN: 978-0-596-00712-6.

# About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.