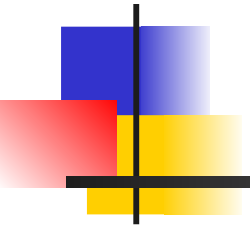


Introduction to Objects

James Brucker

What is an Object?

An object is a program element that *encapsulates* both data and behavior.

An object contains both data and methods that operate on the data.

Objects

Objects represent "things" in the world or problem domain.

Examples:

money

coin purse

a person

game-board

game-piece

a sale

product

Other Use for Objects

We also use objects for constructs or services in our program

Examples:

Math (provides services)

a factory (creates other objects)

event listener

button (in a GUI)

Characteristics of Objects

Objects have

behavior - what an object can do

knowledge or state - what an object knows,

or other objects it knows

identity - two objects are unique, even if they have the same state and knowledge

Objects

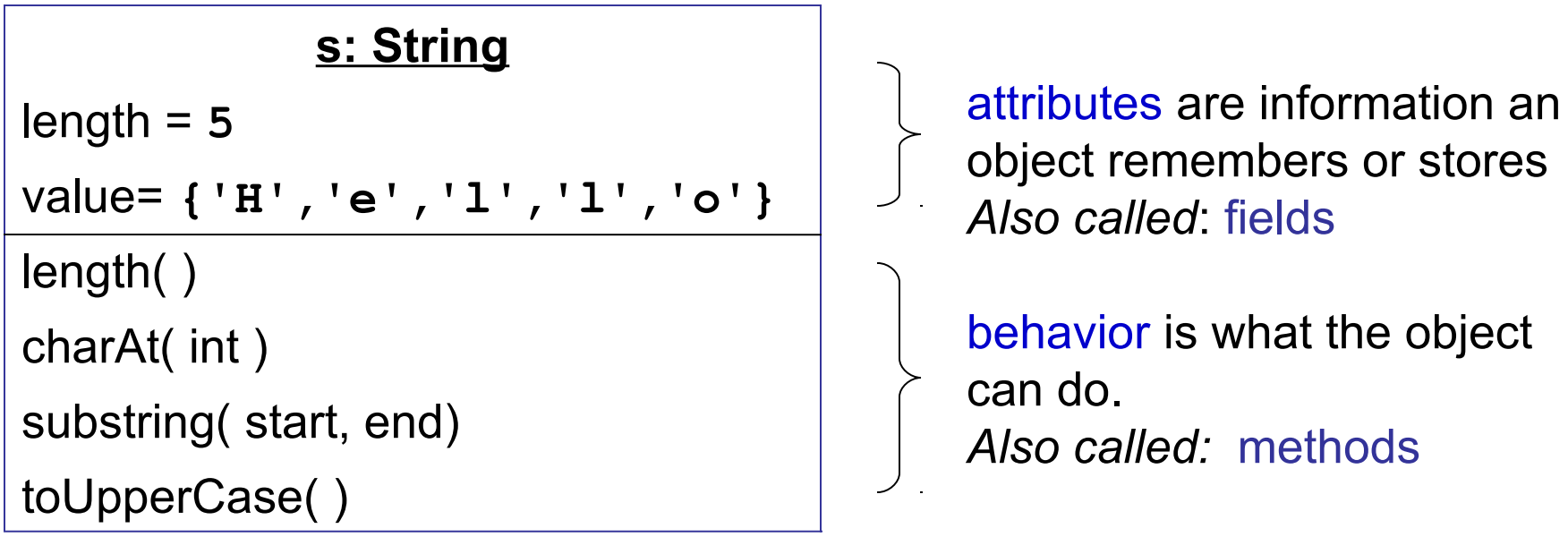
An object has:

attributes - what an object ***knows***

behavior - what can object can ***do***

Consider a String object:

```
String s = "Hello";
```



Objects have Behavior

To invoke an object's behavior, write:

A variable that *refers* to the object

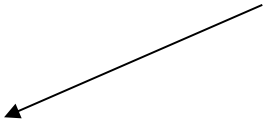
A method that belongs to the object

```
> String s = "Hello Dog";
> s.length()
9
> s.toUpperCase()
"HELLO DOG"
> s.substring(0,5)     // method with a parameter
"Hello"
```

Where Do Objects Come From?

- A **class** defines a **kind of object**.
- **String** is a class.

Memorize this.



Definition:

"A **class** is a **blueprint** or **definition** for a ***kind*** of object**."**

Where is the **`String`** class?

String is part of the Java API.

String class is in the package **`java.lang`**

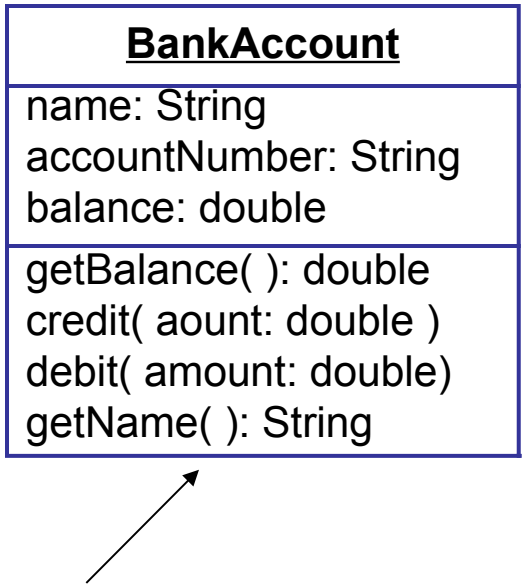
Class

A class is a blue print or design for a kind of object.

A class defines the *attributes* and *behavior* that it will support.

Example:

class name:	**BankAccount**
attributes:	accountNumber, name, balance
behavior:	getBalance( ), credit(amount), debit(amount), getName( )



A UML class diagram - Chapter 3 of UML Distilled

Object

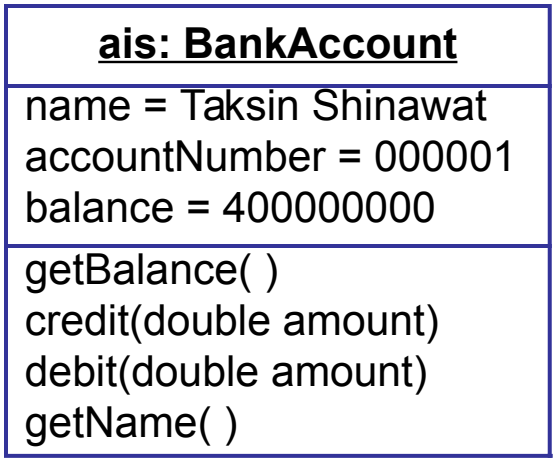
An object is an actual *instance* of the class.

Every object possesses all the attributes and behavior of the class.

Each object has its own set of attribute values, whose value may (and will) differ from other objects.

Example:

```
BankAccount ais =
        new BankAccount("Taksin Shinawat" );
ais.credit( 2000000000 );
ais.credit( 1000000000 );
ais.credit( 1000000000 );
ais.getBalance( ) ;  // = 4,000,000,000
```



A UML object diagram

Objects are *instances* of a Class

To create an object from a class, use "new":

```
Date now =  new Date( );                  // today
Date newyear = new Date(100, 0, 1); // 2000 Jan 1
```

Objects are distinct!

Each object you create using "new" is different.

```
Date now1 = new Date(100, 0, 1 );
Date now2 = new Date(100, 0, 1 );
if (now1 == now2) /* same object */;
```

FALSE

3 Characteristics of Objects

Objects have:

1. **state** - the current condition of an object
2. **behavior** - the actions or behavior an object can perform
3. **identity** - every object is distinguishable, even if two objects have the same *attribute values (state)*

What is identity?

Primitive values don't have identity.

```
int n = 10;
int m = 10;
n == m   // true - they are the same value
```

But objects are unique, even if their states are the same

```
Integer a = new Integer(10);
Integer b = new Integer(10);
a == b   // false - a and b refer to unique objects
```

Changing an Object's State

Some methods **change** an object's state (attributes).

These are *usually* "set" methods.

Change a date:

```
Date now =  new Date( );          // today
System.out.println( now );        // maybe 16 Jan 2013
now.setMonth( 11 );               // change to Dec.
now.setDate( 1 );                 // 1st day of month
now.setHour( 12 );                // 12:00 noon
System.out.println( now );        // value has changed
```

More about Creating Objects

1. Use "new" to create an object from a Class.

```
Coin fivebaht = new Coin(5);
```

fivebaht *is a* **reference** to a Coin object (like a pointer)

2. Some classes have a *factory method* for creating objects.

```
Calendar cal = Calendar.getInstance( );
```

getInstance( ) *is a static method that creates a new Calendar object*

Factory Method

To create a Calendar use getInstance( )

```
Calendar cal = Calendar.getInstance( );
```

java.util.Calendar does not have a public constructor.

- So, you can not write "new Calendar( )".
- Instead you use a *static method* named **`getInstance`**( ).

Reasons of this:

- Enable data validation before the object is created.
- Enable polymorphism: a factory method can return any compatible type, not just the declared type (Calendar).

Calendar uses a Factory Method

To create a Calendar use getInstance( )

```
Calendar cal = Calendar.getInstance( );
```

java.util.Calendar does not have a public constructor.

- So, you can't create a Calendar object using "new Calendar( )".
- Instead you use a **Factory Method** named `getInstance( )`.

Advantages of this approach:

- Enables data validation before the object is created.
- Supports polymorphism: constructor always creates the named object type. A factory method can return any compatible type.

The Meaning of "Identity"

Objects have 3 characteristics: state, behavior, and identity.

Identity means that objects are distinct entities.

Identity does not mean a *variable* that refers to an object.

a variable is just a "reference" (pointer) to an object

a variable is not part of the object

Object Identity

Objects are distinct even if their attributes are the same:

```
Student s1 = new Student( "Harry Hacker", 1111 );
Student s2 = new Student( "Harry Hacker", 1111 );
System.out.println ( s1 == s2 );  // FALSE
```

- Compare this with primitive data types *(value types)*:

```
int s1 = 1111;
int s2 = 1111;
System.out.println ( s1 == s2 ); // TRUE
```

Variables *refer* to Objects

```
String s;
```

Defines a variable named "s" of type String. It doesn't *refer* to any String object yet, so its value is null.

```
s = "Hello";
```

Make s refer to a String object "Hello"

```
s = new
String("Bye");
```

Make s refer to a String object "Bye".

```
String s2 = s;
```

Define another String variable s2, and make it *refer* to the object ("Bye") that s refers to.

This does not copy the object!

A Variable is NOT an Object

```
String s;
```
s is NOT a String object

```
Date now;
```
now is NOT a Date object.

```
s = "hello";
```
s is *still* NOT a String.

s *refers to a String* object.

```
now = new Date( );
```
now is *still* NOT a Date.

now *refers to a Date object.*

Define Your Own Class

- This is covered in my "Java Basics" slides named Introduction-to-Java-2

Review

1. What is the definition of a **class**?

2. What are the **3 characteristics of objects**?

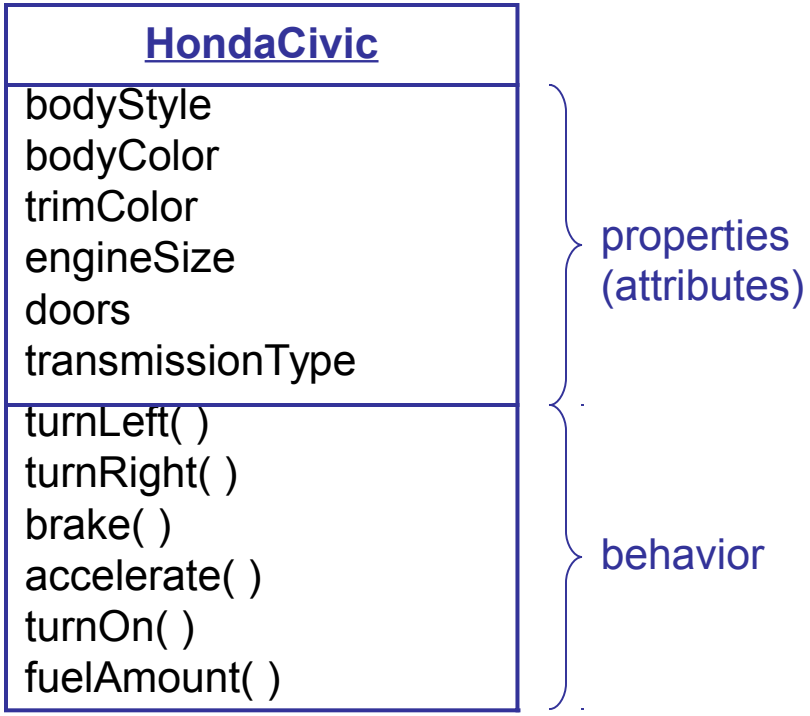
3. How do you create a Date object for the date Feb 15, 2000?

4. Is this true or false? Why?

```
Double x = new Double(1.0);
Double y = new Double(1.0);
(x == y)
```

HondaCivic class

- A Honda Civic owner doesn't need to know **these details** (how a Civic works).
- He only needs to know a Honda Civic's ***properties*** (public attributes) and ***behavior***.
- For example:



Buying A Honda Civic

- You go to the Honda dealer and say...

Buying A Honda Civic (2)

Dealer offers you the *class* (definition) for a Honda Civic...

Construction and operation of a Honda Civic: complete documents.

Buying A Honda Civic (3)

but you can't drive blue prints and documents

That's not exactly what I want.
I want a **car** I can **drive**...

I see... you want an ***instance*** of Honda Civic -- a Honda Civic ***object***.

Object Identity Example

- Two Honda Civic cars can be distinguished even if they exactly the same features and same state (brand new)

!=