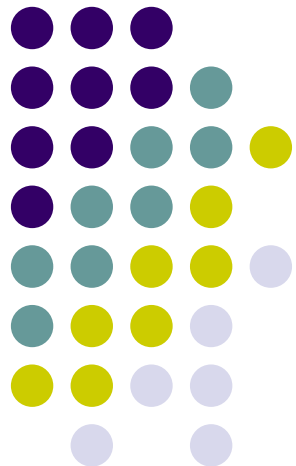
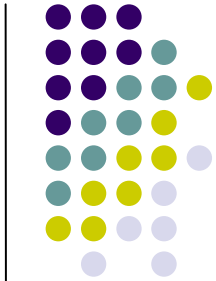
Classes & Interfaces

Java’s Object Oriented System

Justin Mazzola Paluska



Keywords



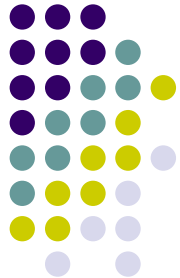
- **Class** – a template of a data object
- **Interface** – a specification
- **Instance** – an instantiation of a Class or Interface physically represented in memory

- **Method** – a set sequence of instructions

- **Instance Field** – variable associated with a particular instance.
- **Static Field** – variable shared among all instances of a Class

class members

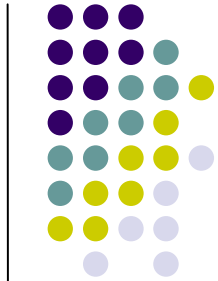
Data Types



- There are two types in Java
  - Primitive types
  - Reference types

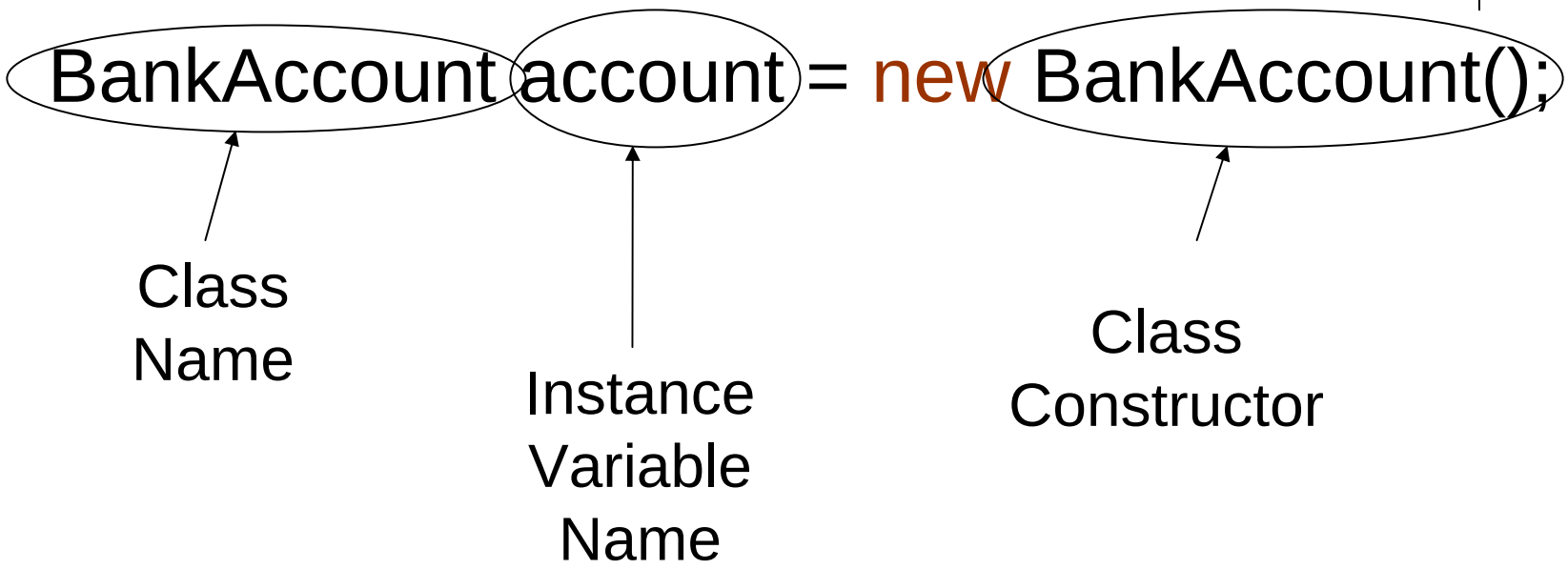
- Most of your time is spent using Reference types.

Reference Types

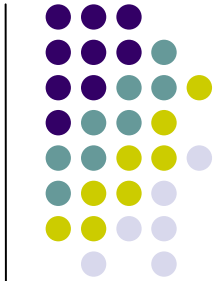


- Also known as Objects
- To create an **instance** of a reference type, use the new keyword in Java
- The new keyword:
  1. Makes space for the new object in memory
  2. Calls the constructor you specify
  3. Returns a reference to the new object

Example Instantiation of a Class

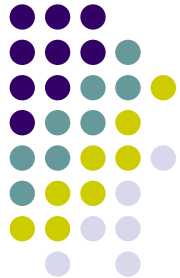


Use of instances



- Call methods off of instances:
  - account.withdraw(amount);
  - account.deposit(amount);
- Access its instance variables:
  - account.id
  - account.balance
- When we're done with an object, we just stop using it.
  - Java will garbage collect the object when there are no more references to it.

Defining a Class

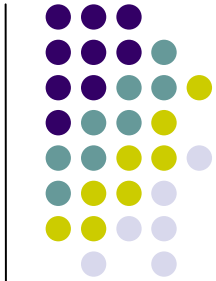


- The template for a class definition follows:

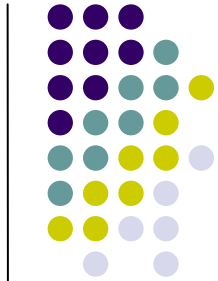
```
[access] [abstract/final] class className
    [extends superClassName]
    [implements interfaceNames…] {
    //constructors
    //member functions
    //member variables
}
```

Simple Example

```
public class BankAccount {
    ...
}
```

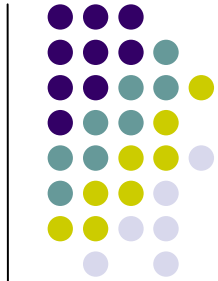


Class Members



- In class definitions we can define the following members:
  - Constructors
  - Instance and static methods
  - Instance and static fields
  - Nested classes

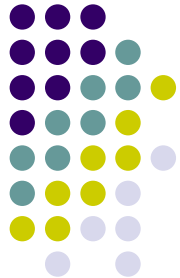
Constructors



- Must have the same name of the Class that they are in
- Can have multiple constructors per Class
- Handles initialization of your class
- Template:

```
[access] className ([arguments…]) {
    //constructor body
}
```

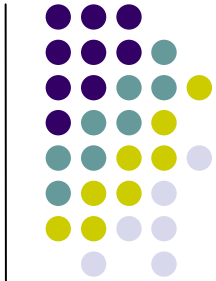
Example: Single Constructor



```
public class BankAccount {
    public BankAccount () {
        ...
    }
}
```

Notice that the name of the constructor is the same as the class

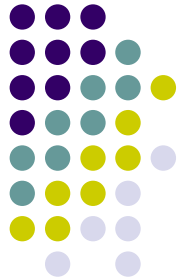
Example: Multiple Constructors



```
public class BankAccount {
    public BankAccount () {
        …
    }
    public BankAccount (int initialAmount) {
        …
    }
}
```

These are different constructors because they take in different arguments

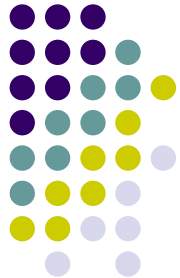
Methods



- Methods perform functions
- Methods work on the state of the class
- Like Scheme, methods can take in multiple arguments, and return up to one value
- If no value is to be returned, use the keyword void
- A class can have as many methods as needed
- Template:

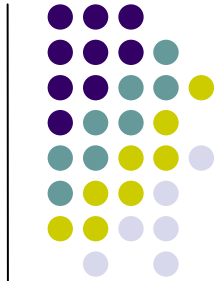
```
[access] returnType methodName ([arguments…]) {
        //method body
}
```

Example Methods



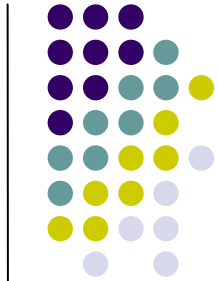
```
public class BankAccount {
    public void withdraw (int amount) {
        …
    }
    public int getAmount () {
        …
    }
}
```

Method Overloading



- A class can have two functions with the same name in a class as long as their arguments differ.
- Example:
  - void foo () {…}
  - void foo (int bar) {…}
- Java knows which method to call based on the method signature
- Example: myClass.foo(7) //calls 2nd method

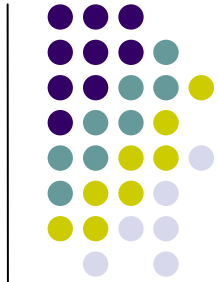
Fields



- A field is like a variable, it stores state
- A field has a associated data type which determines the type of data that this field will hold
- Template:

  [*access*] *dataType fieldName* [= *value*];

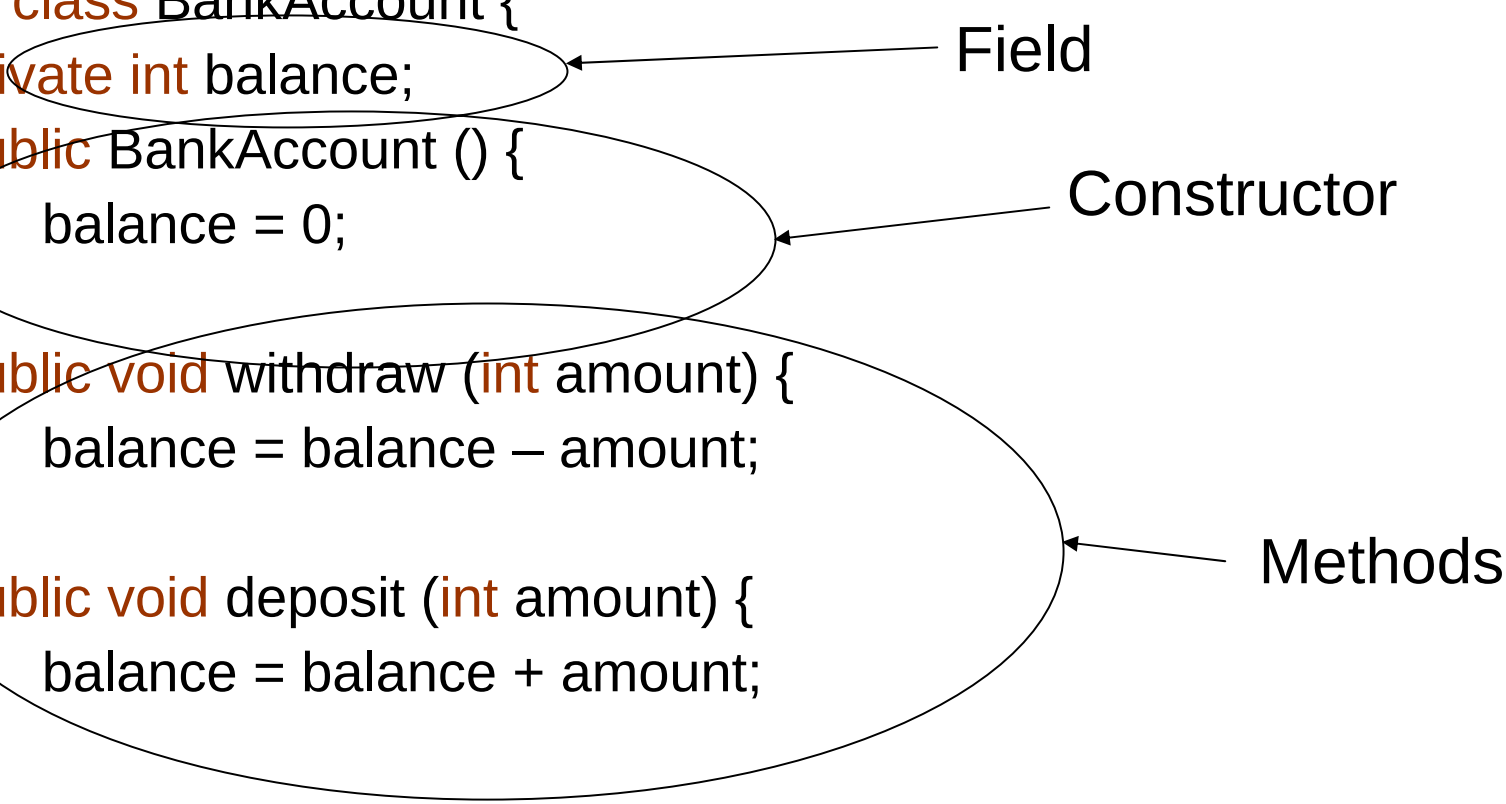
Example Fields



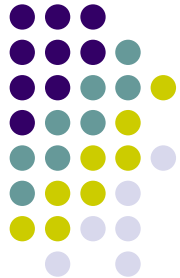
```
public class BankAccount {
    public int balance;
    public Date lastWithdrawal;
    public List transactions;
}
```

Bringing It Together

```
public class BankAccount {
    private int balance;
    public BankAccount () {
        balance = 0;
    }
    public void withdraw (int amount) {
        balance = balance – amount;
    }
    public void deposit (int amount) {
        balance = balance + amount;
    }
}
```

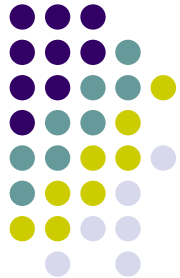


Accessors



- Before we saw the placeholder [access].
- There are 4 types of access keywords to describe which classes have access:
  - public – any other class in any package
  - protected – any subclass has access
  - (default) – only classes within the same package
  - private – only accessible from within a class
- Good for keeping data abstraction intact

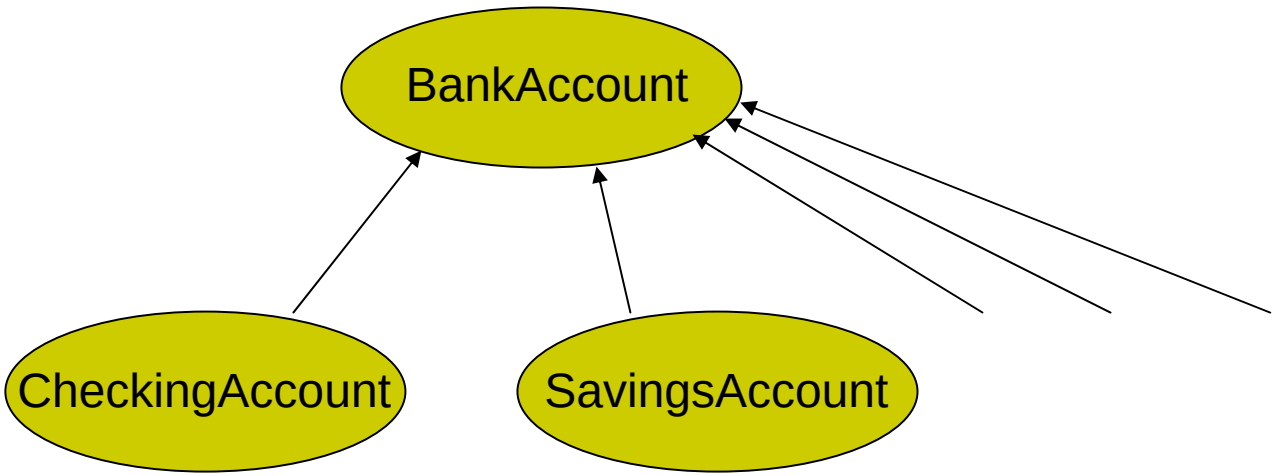
Inheritance



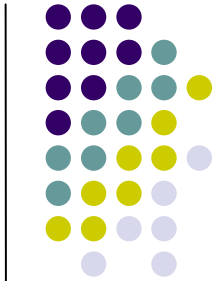
- Allows classes to inherit functionality from other classes
- Allows data and procedural abstraction
- Decreases complexity of large software systems

Checking and Savings

- Two separate ideas with different behaviors, but there exists overlap of functionality



Interfaces

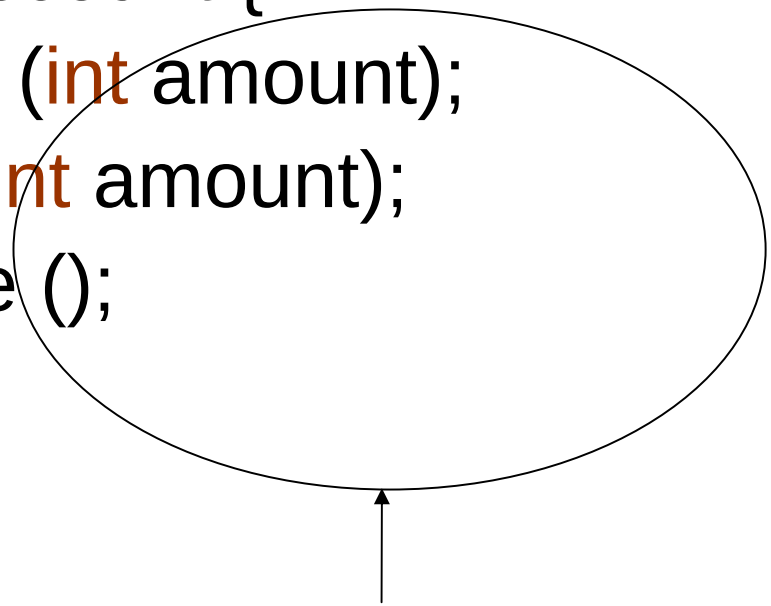


- An interface is a specification of a Class
- Declares methods but does not define them
- Interfaces do not have constructors
- Template:

```
[access] interface interfaceName
        [extends interfaceNameList…] {
 //method declarations
}
```

Example Interface

```
public interface BankAccount {
    public void withdraw (int amount);
    public void deposit (int amount);
    public int getBalance ();
}
```



Notice that for method declarations, the method body is not defined.

How do we use the Interface?

- We make classes or other interface **implement** or **extend** the interface.
- If a class implements an interface, that class must provide an implementation (a method body) for every method specified by the interface
  - If a class implements multiple interfaces, it must implement all methods of every interface it chooses to implement

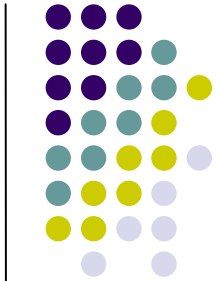
Example Interface Use

```
public class CheckingAccount implements BankAccount {
        private int balance;
        public CheckingAccount (int initial) {
                balance = initial;
        }

        //implemented methods from BankAccount
        public void withdraw (int amount) {
                balance = balance – amount;
        }
        public void deposit (int amount) {
                balance = balance + amount;
        }
        public int getBalance () {
                return balance;
        }
}
```

Since CheckingAccount implements BankAccount, it must provide implementations for these methods

Abstract Classes



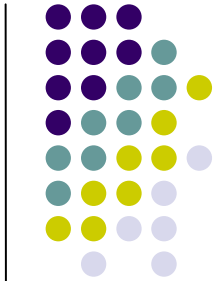
- Abstract classes are a mix between interfaces and classes
  - can have defined method bodies
  - can have fields
- Helps to capture the idea of state as well as functionality
- Template:

  See Class template (use keyword abstract)

Advantage of Abstract Classes

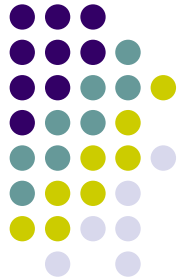
- For our BankAccount example we can choose to provide implementations for methods we know is common, and declarations for methods that might differ
- Let’s build an abstract class for BankAccount

Example: Abstract Class



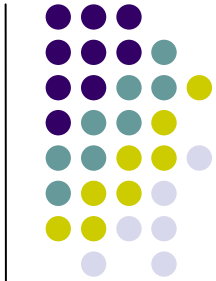
```
public abstract class BankAccount {
    protected int balance;
    public int getBalance () {
        return balance;
    }
    public void deposit (int amount) {
        balance = balance + amount;
    }
    public void withdraw (int amount);
}
```

Example:
Class Extension



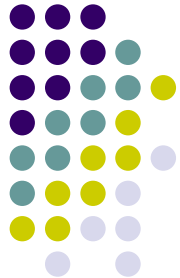
```
public class CheckingAccount extends BankAccount {
    public CheckingAccount () {
        balance = 0;
    }
    public void withdraw (int amount) {
        balance = balance – amount;
    }
}
```

Example: Class Extension



```
public class SavingsAccount extends BankAccount {
    private int numberOfWithdrawals;
    public SavingsAccount () {
            balance = 0;
      numberOfWithdrawals = 0;
    }
    public void withdraw (int amount) {
      if (numberOfWithdrawals > 5) {
                throw new RuntimeException (“Cannot make >5 withdrawals a month”);

      } else {
                balance = balance – amount;
                numberOfWithdrawals++;
      }
    }
    public void resetNumOfWithdrawals () {…}
}
```



Break