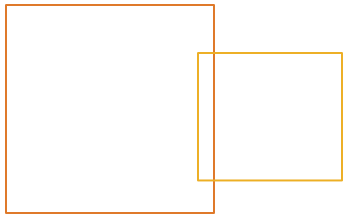
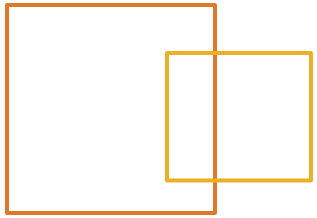
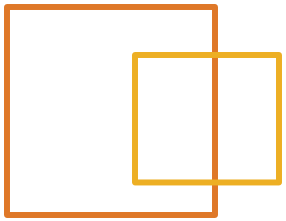
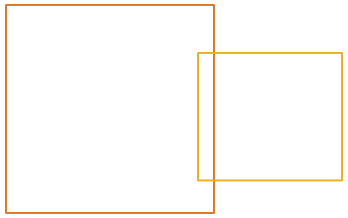
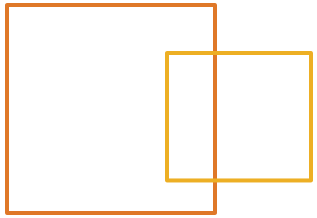
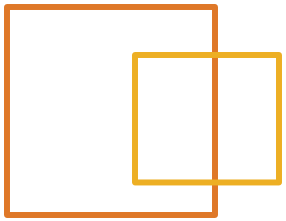





Fast Track to Java

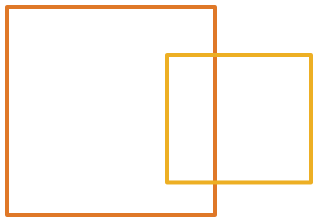
Customized for Starbucks

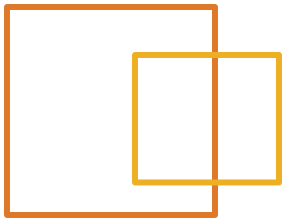
Delivered by DevelopIntelligence


Inheritance in Java

Objectives





- At the end of this module, you should be able to:
- Describe the OO concepts of abstraction and inheritance
- Implement inheritance and method over-riding
- Use the protected keyword
- Use abstract methods and classes
- Use interfaces
- Use up-casting and down-casting

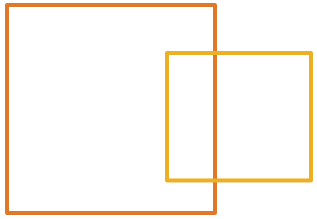
© DevelopIntelligence http://www.DevelopIntelligence.com

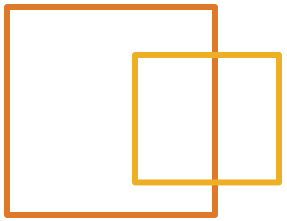
Abstraction the Real World - Concrete Classes

- Every object is of some type. This is how we naturally think about the world, it allows efficient information processing
  - `String s1 = new String();`
  - `MyType mt = new MyType();`
- Types are defined by a process of abstraction or generalization - grouping a collection of objects together based on some common features; and creating a prototype

© DevelopIntelligence http://www.DevelopIntelligence.com

Abstraction



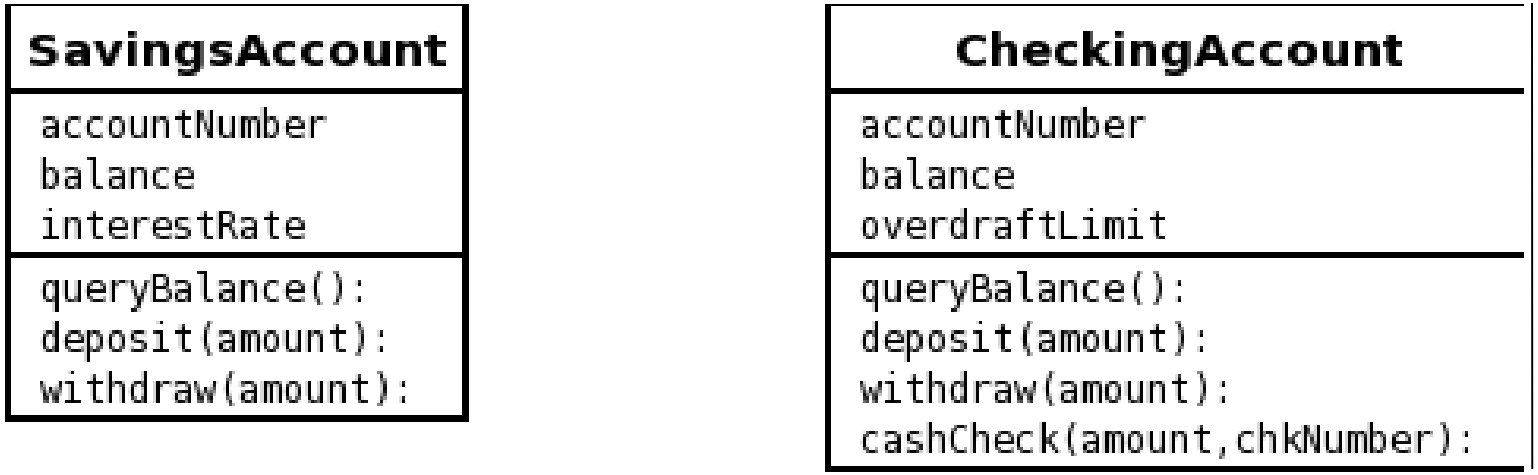


- Types exist only because the observers define them
- During the design process all types in the problem domain are synthesized down to a set of design classes that are implemented in code
- No "right" collection of classes exist
    - The collection must address the business problem
    - Some classes are for design or implementation reasons
    - Different sets display different benefits, especially during maintenance
    - Your designs probably get more elegant with experience

© DevelopIntelligence http://www.DevelopIntelligence.com

Inheritance in the Real World

Look at our banking example



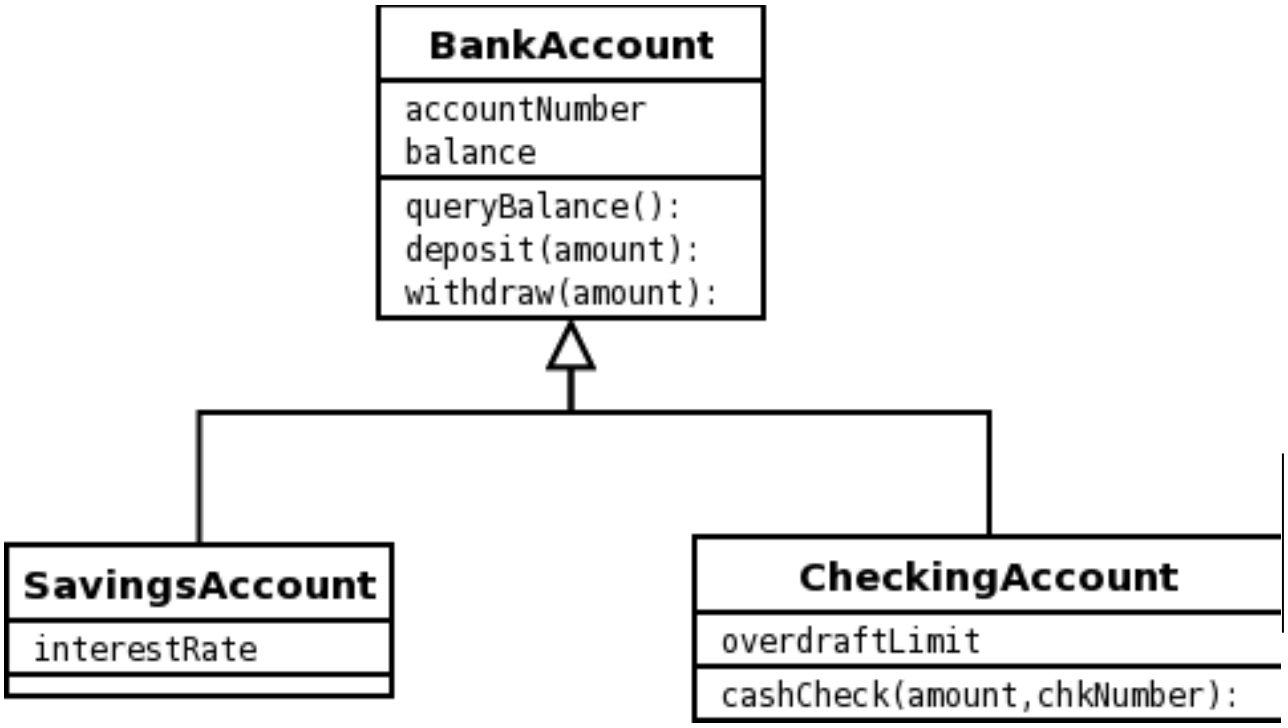
The two bank account types

When doing abstraction analysis, look for

- common attributes
- common behaviors

© DevelopIntelligence http://www.DevelopIntelligence.com

Abstraction and Inheritance



Abstract BankAccount

Notice where the commonality has been placed, in another class!

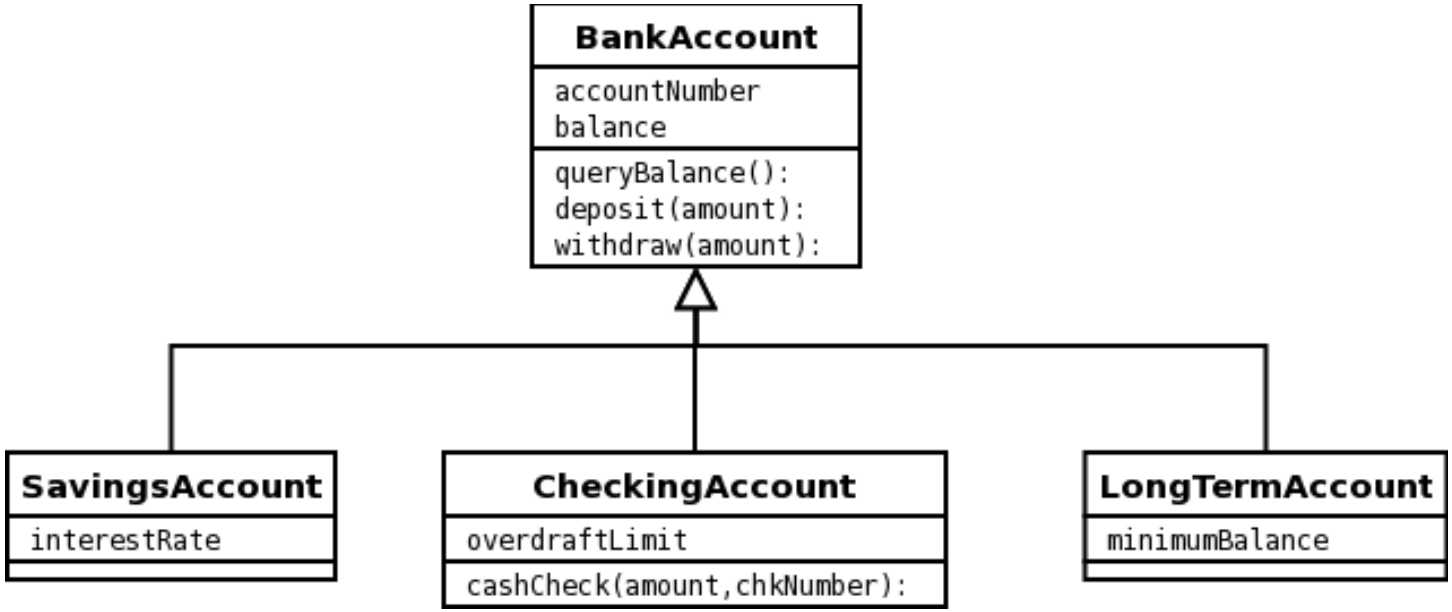
© DevelopIntelligence http://www.DevelopIntelligence.com

Abstraction and Inheritance (cont.)

- "What is the relationship between personal customers and checking accounts?”
- Reason by inheritance - Customers have accounts therefore, personal customers can have accounts
- Checking account is a kind of account, it inherits the property of "being held" by a customer

© DevelopIntelligence http://www.DevelopIntelligence.com

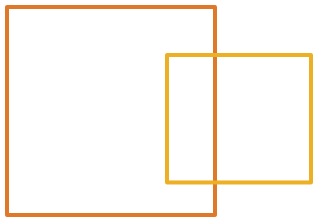
Abstraction and Inheritance (cont.)

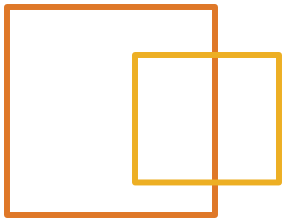


Adding a LongTermAccount

© DevelopIntelligence http://www.DevelopIntelligence.com

Some Jargon





- The BankAccount class is called
  - Base class, super class, parent class, or
  - The “generalization”
- The SavingsAccount and CheckingAccount classes are called
  - Derived classes, subclasses, child class, or
  - “Specializations”
- Different people have different terms which can be somewhat confusing

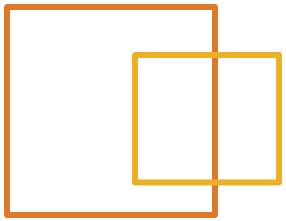
© DevelopIntelligence http://www.DevelopIntelligence.com

Inheritance in Java

- Java only supports single implementation inheritance
  - A class can have only one parent
  - Follows along the lines of scientific classification
  - This is very different from C++ and C#
  - Easier to manage
  - Less error prone
- Inheritance in Java uses the extends keyword
  class Child extends Parent

© DevelopIntelligence http://www.DevelopIntelligence.com

Inheritance in Java (cont.)



- Inheritance in Java refers to
  - Instance variables
  - Instance methods
- Constructors are not inherited
- You can have control over what variables and methods are inherited using access modifiers
  - private variables and methods are not inherited
  - public, protected, and default are, at some level
- static variables and methods are not inherited

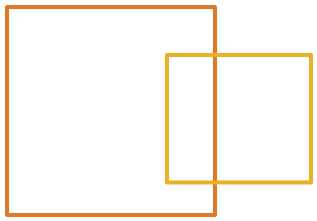
© DevelopIntelligence http://www.DevelopIntelligence.com

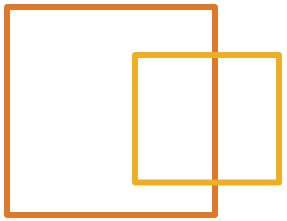
Inheritance Example

```
// Superclass BankAccount
class BankAccount {
  float balance;
  String accountNumber;
  float queryBalance() {/*  code */
  float deposit(float amount) { /* code */}
  float withdraw(float amount) {/* code */}
}
// Subclass SavingsAccount
class SavingsAccount extends BankAccount {
  float interestRate;
}
// Subclass CheckingAccount
class CheckingAccount extends BankAccount {
  float overdraftLimit;
  float cashCheck(float amount, int ChkNumber) {
    /* code */
  }
}
```

© DevelopIntelligence http://www.DevelopIntelligence.com

Final Classes





- Declare a class to be final using the final keyword preceding the class definition.
- This prohibits the class from being used as the base class in any inheritance structure
- The reasons for declaring a class final is always because of a design issue in the application
- For example, we may declare a class final because having subclasses would allow objects to circumvent business rules or security checks

© DevelopIntelligence http://www.DevelopIntelligence.com

Final Class Example

```
// Superclass BankAccount is now final
// This will NOT compile
final class BankAccount {
  float balance;
  String accountNumber;
  float queryBalance() {/*  code */
  float deposit(float amount) { /* code */}
  float withdraw(float amount) {/* code */}
}
// Subclass SavingsAccount
class SavingsAccount extends BankAccount {
  float interestRate;
}
// Subclass CheckingAccount
class CheckingAccount extends BankAccount {
  float overdraftLimit;
  float cashCheck(float amount, int ChkNumber) {
    /* code */
  }
}
```

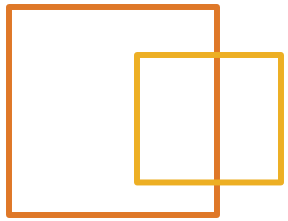
© DevelopIntelligence http://www.DevelopIntelligence.com

Private Variables Inheritance Example

```
// balance is private
// This will NOT compile
class BankAccount {
  private float balance;
  /* -- more code -- */
}
// Subclass SavingsAccount accesses private variable
// balance in super class
class SavingsAccount extends BankAccount {
  public queryBalance() {
    return balance;
  }
}
```

© DevelopIntelligence http://www.DevelopIntelligence.com

Protected Access Modifier



- Use of the protected keyword for instance variables may not be desirable
    - protected variables and methods are inherited
    - protected variables and methods can also be accessed by other objects of other types
- For strict design correctness, the private keyword suffices
- If you need more “protection” than protected but not as restrictive as private , consider using the default access modifier
    - Inherited by subclasses in the same package
    - Accessed by classes in the same package

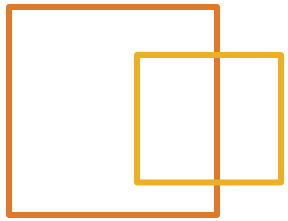
© DevelopIntelligence http://www.DevelopIntelligence.com

Protected Access Modifier Inheritance Example

```
// balance is protected - okay now
class BankAccount {
  protected float balance;
   /* -- more code -- */
}
// Sub class SavingsAccount can access
// protected variable balance in super class
class SavingsAccount extends BankAccount {
  public queryBalance() {
    return balance;
  }
}
```

© DevelopIntelligence http://www.DevelopIntelligence.com

Implementing Inheritance



- Sometimes it is useful for a child to change an inherited behavior
- This can be performed using method over-riding
- Instead of the inherited method being invoked, the over-ridden method will be invoked
- In Java, the *type of the object* determines the behavior you get, *not the type of the reference*
    - Called virtual invocation or late binding

© DevelopIntelligence http://www.DevelopIntelligence.com

Implementing Inheritance Example

```
class Parent {
  private String priVar = "(Parent Private)";
  protected String proVar = "(Parent Protected)";
  public  String pubVar = "(Parent Public)";
  public void meth() {
    System.out.println("(Parent method)");
    System.out.println(priVar+"  "+pubVar+" "+proVar);
  }
  public void methPar() {
    System.out.println("Parent method");
    System.out.println(priVar+"  "+pubVar+" "+proVar);
  }
}
class Child extends Parent {
  public  String  pubVar = "(Child Public)";
  private String  priVar = "(Child Private)";
  @Override // Ask the compiler to check we spelled it right
  public void meth() { //over-ridden method
    System.out.println("Child method");
    System.out.println(priVar+"  "+pubVar+" "+proVar);
 }
```

Implementing Inheritence Example (cont.)

```
  public static void main(String [] args) {
    Child c = new Child();
    c.meth();
    c.methPar();
  }
  /* Output is
   * Child method
   *(Child Private)   (Child Public)  (Parent Protected)
   * Parent method
   *(Parent Private)   (Parent Public)  (Parent Protected)
   */
}
```

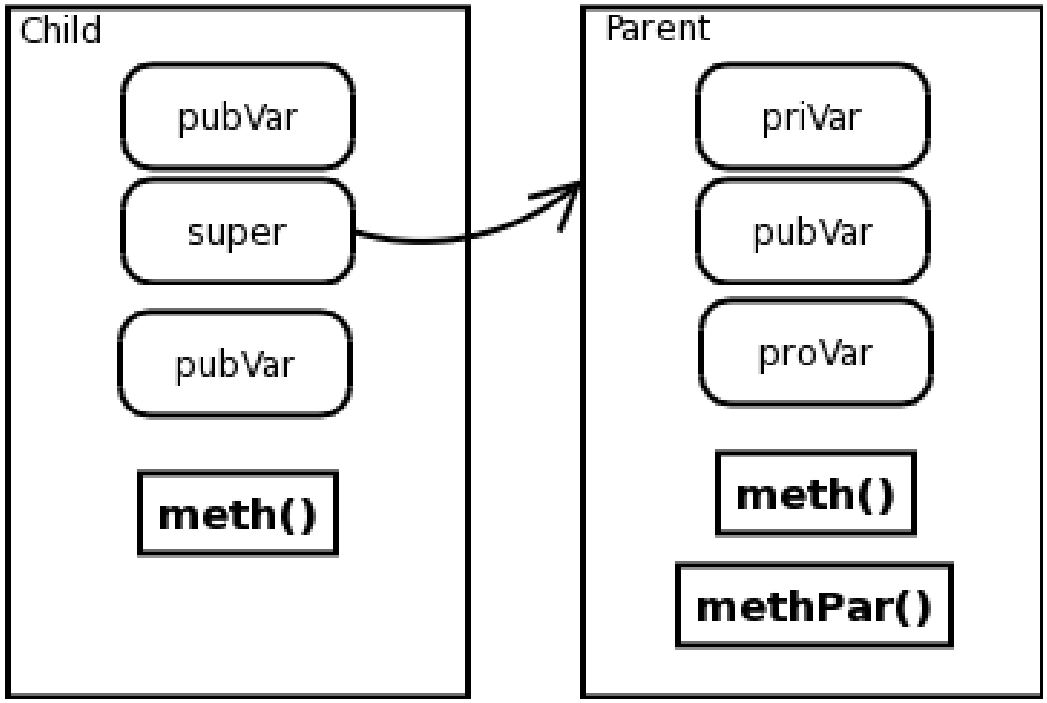
© DevelopIntelligence http://www.DevelopIntelligence.com

More About Implementing Inheritance

- A child might need to interact with its parent
  - This can be performed using a built-in reference ***`super`***
  - Use the dot-notation with ***`super`***
- Typically ***`super`*** is used to access an over-ridden method in the parent class
- It can also be used to explicitly access “shadowed” variables in the parent class
- Parent classes can encapsulate sensitive data and behaviors
  - Mark them ***`private`***
  - This might prevent method over-riding

© DevelopIntelligence http://www.DevelopIntelligence.com

More About Implementing Inheritance (cont.)



Result of inheritance from example

© DevelopIntelligence http://www.DevelopIntelligence.com

Shadowing Example

```
class Parent {
  /* just like example
}
class Child extends Parent {
  public  String  pubVar ="(Child Public)";
  private String  priVar = "(Child Private)";
  public void meth() {
    System.out.println("Child method");
    System.out.println(priVar+ "  " +
                       super.pubVar + " " + proVar);
  }
  public static void main(String [] args) {
    Child c = new Child();
    c.meth();
    c.methPar();
  }
  /* Output is
   * Child method
   * Child Private)   (Parent Public)  (Parent Protected)
   * Parent method
   * Parent Private)   (Parent Public)  (Parent Protected)
   */
}
```

© DevelopIntelligence http://www.DevelopIntelligence.com

Invoking A Parent Method Example

```
class Parent {
  /* just like example
}
class Child extends Parent {
  public  String  pubVar ="(Child Public)";
  private String  priVar = "(Child Private)";
  public void meth() {
    System.out.println("Child method");
    System.out.println(priVar+"  "+super.pubVar+" "+proVar);
  }
  public void up() {
    System.out.println("Up method");
    super.meth();
  }
  public static void main(String [] args) {
    Child c = new Child();
    c.up();
  }
  /* Up method
   * Parent method)
   * Parent Private)  (Parent Public) (Parent Protected)
   */
}
```

© DevelopIntelligence http://www.DevelopIntelligence.com

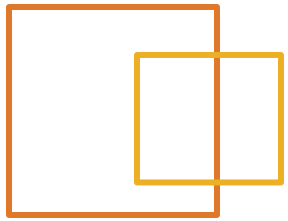
Extending and Implementing Methods

- When over-riding a method
- The method signature in the child class can have the same access modifier
- The method signature in the child class can be less restrictive

```
class Parent {
  protected void m1() {}
  public void m2() {}
}

class Child extends Parent {
  // this is allowed public is less restrictive
  public void m1() {}
  // not allowed, must be public
  protected void m2() {}
}
```

© DevelopIntelligence http://www.DevelopIntelligence.com

Accessing Super Class



- Objects are initialized using constructors
- JVM calls the constructor for each super class
- By default, the JVM will call the default or no-argument constructor in the super classes when creating the child object
- In some cases, the creation of the child object will require a different constructor be called in the parent class
    - Use the ***`super`*** keyword in a manner similar to the use of the this keyword in the constructor
    - ***`super`*** must be the first execution in the constructor

© DevelopIntelligence http://www.DevelopIntelligence.com

Accessing Parent Constructors Example

```
class Parent {
  Parent() {
    System.out.println("Parent()");
  }
  Parent(int i) {
    System.out.println("Parent(int)");
  }
}
class Child extends Parent {
  Child () {
    super(); //redundant.. Default constructor would be called anyway
    System.out.println("Child()");
  }
  Child (int i) {
    super(i);
    System.out.println("Child(int i)");
  }
  Child (int i, int j) {
    super(i);
    System.out.println("Child(int i, int j)");
  }
```

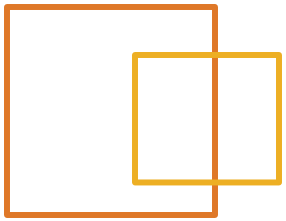
© DevelopIntelligence http://www.DevelopIntelligence.com

Accessing Parent Constructors Example (cont.)

```
  public static void main(String [] args) {
    Child c1 = new Child();
    Child c2 = new Child(1);
    Child c3 = new Child(1,2);
  }
} //end of Child class
  /* This produces the output
   * Parent()
   * Child()
   * Parent(int)
   * Child(int i)
   * Parent(int)
   * Child(int i, int j)
   */
```

© DevelopIntelligence http://www.DevelopIntelligence.com

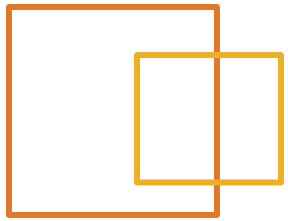
Abstract Classes



- We have already looked at creating generalizations and specializations of types
    - Created concrete base class
    - Created concrete child class
- Another way, and possibly more common way, to create generalizations and specializations of types is with an abstract class
    - Provide description of the required functionality for all specializations
    - Need not provide all/any of the implementation
    - Implementation can be provided by specialization

© DevelopIntelligence http://www.DevelopIntelligence.com

Abstract Classes (cont.)



- Abstract classes
  - Are partially defined, partially undefined where as concrete classes are fully defined
  - Can not be instantiated
  - Must be extended to become fully defined
- Abstract classes can contain
  - Instance variables and methods
  - Class variables and methods
  - Constructors
  - Abstract methods

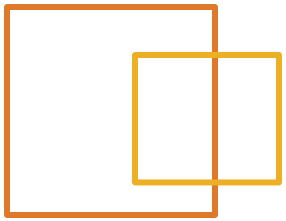
© DevelopIntelligence http://www.DevelopIntelligence.com

Abstract Classes Example

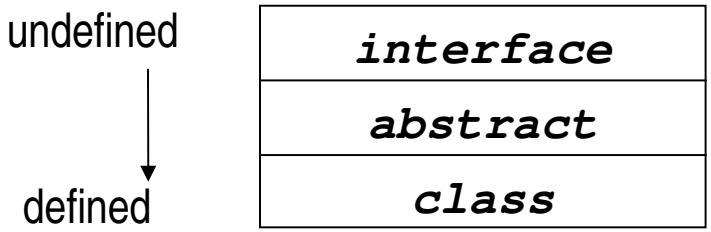
```
abstract class Teller {
  // regular method, useless implementation
  public String deposit(String amt, String act)
    return "Not implemented";
  }
  // abstract method, no implementation code
  public abstract String withdraw(String amt, String act);
}

class HumanTeller extends Teller {
  public String withdraw(String amt, String act) {
    return "success";
  }
  public static void main(String [] args) {
    Teller t = new Teller(); // Compiler error!!!
    // but this works:
    HumanTeller ht = new HumanTeller();
  }
}
```

© DevelopIntelligence http://www.DevelopIntelligence.com

“Multiple” Inheritance

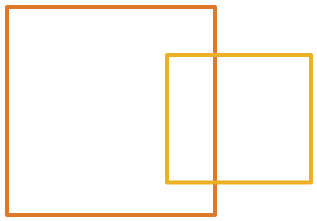


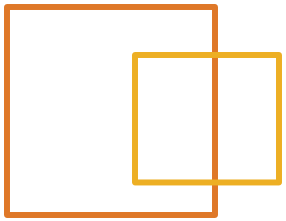
- Java allows a form of multiple inheritance
  - Objects may implement behaviors of multiple interfaces
  - Not true multiple inheritance
- Interfaces are totally abstract classes
  - No method bodies
  - No member variables
  - But does allow “constants”



© DevelopIntelligence http://www.DevelopIntelligence.com

Interfaces





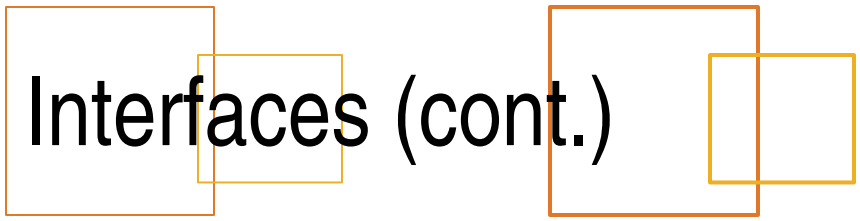
- interfaces are defined in .java files
  - Structure similar to classes

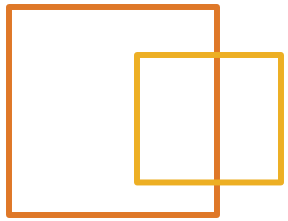
```
[public] interface <interface_name> {
    […]
}
```

- interfaces may be empty or may contain
  - public abstract method declarations
  - public static final variables (constants)
- Abstract methods are
  - Instance, *not class*, methods without a body
  - `[public] abstract void doSomething();`

© DevelopIntelligence http://www.DevelopIntelligence.com

Interfaces (cont.)





- A class can take on the behavior of an interface by
  - Implementing the interface, using ***implements***
  - Defining all of the inherited abstract methods
- Classes can implement many interfaces

```
class Child implements Type1, Type2,
  Type3
```

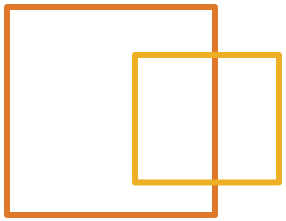
© DevelopIntelligence http://www.DevelopIntelligence.com

Interface Example

```
public interface LedgerAccount {
  public abstract String credit(String amt);
  public abstract String debit(String amt);
}

public class BankAccount {
  public String deposit(String amt) { /*…*/ }
  public String withdraw(String amt) { /*…*/ }
}

public class SavingsAccount extends BankAccount
  implements LedgerAccount {
  public String credit(String amt) { /*…*/ }
  public String debit(String amt) { /*…*/ }
}

public class LoanAccount implements LedgerAccount {
  public String credit(String amt) { /*…*/ }
  public String debit(String amt) { /*…*/ }
}
```

© DevelopIntelligence http://www.DevelopIntelligence.com

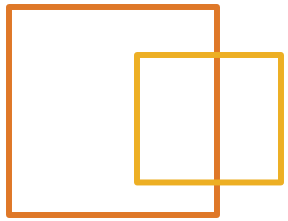
Multiple Interface Example



```
public interface LedgerAccount {
   public abstract String credit(String amt);
   public abstract String debit(String amt);
}

public interface Persistent {
   public abstract void read(String url);
   public abstract void write(String(url);
}

public class SavingsAccount extends BankAccount
  implements LedgerAccount, Persistent {
  public abstract String credit(String amt) { /* … */ }
  public abstract String debit(String amt) { /* … */ }
  public abstract void read(String url) { /* … */ }
  public abstract void write(String(url) { /* … */ }
}
```

© DevelopIntelligence http://www.DevelopIntelligence.com

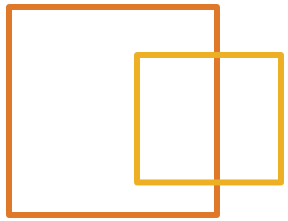
Type Polymorphism



- Polymorphism allows an object to be viewed as different types:
  - SavingsAccount is also:
    - A BankAccount
    - A LegerAccount
    - A Persistent
    - A java.lang.Object

© DevelopIntelligence http://www.DevelopIntelligence.com

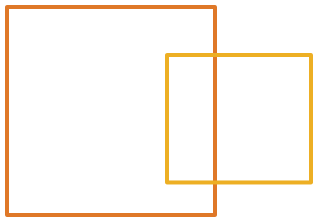
Polymorphism Example

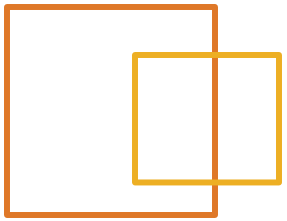


```
interface LedgerAccount {
  /* body code */
}

interface Persistent {
  /* body code */
}

class BankAccount {
  /* body code */
}

class SavingsAccount extends BankAccount implements LedgerAccout,
Persistent {
  /* body code */
}

  // Elsewhere in code…

  SavingsAccount s1 = new SavingsAccount();

  LedgerAccount s2 = new SavingsAccount();

  BankAccount s3 = s1;
```

© DevelopIntelligence http://www.DevelopIntelligence.com

Type Casting

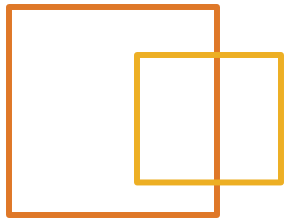




- If an object has more than one type, how do you determine its functionality?
  - Look at the type of the reference variable
  - Utilize “casting”
- Casting converts the type of a reference
  - Might lose access to specific funcionality
  - Reference example
    s1 = (SavingsAccount) s2;
- Java handles widening (up-casting) automatically
  - Narrowing (down-casting) must be performed manually
  - Recall that casting also works with primitives

© DevelopIntelligence http://www.DevelopIntelligence.com

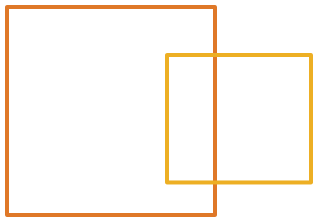
Type Casting Example

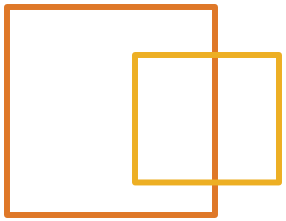


```
interface LedgerAccount {
  /* code */
}
interface Persistent {
  /* code */
}
class SavingsAccount extends BankAccount implements LedgerAccout,
Persistent {
  /* code */
}

 // Elsewhere in code…
 SavingsAccount s1 = new SavingsAccount();
 BankAccount s3 = s1; // this is Okay
 s1 = s3;   //illegal
 s1 = (SavingsAccount) s3;  // now this is okay
 LedgerAccount s4 = new SavingsAccount(); // okay
 s1 = (SavingsAccount) s4;  // need to cast here too
```

© DevelopIntelligence http://www.DevelopIntelligence.com

Summary



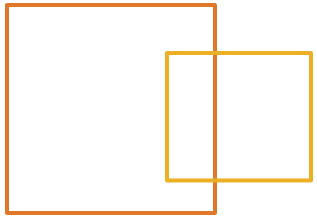


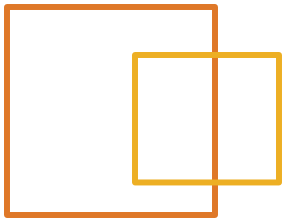
We covered

- The OO concepts of abstraction and inheritance
- Implementing inheritance and over-riding
- Using the protected keyword
- Using abstract methods and classes
- Using interfaces
- Using up-casting and down-casting

© DevelopIntelligence http://www.DevelopIntelligence.com

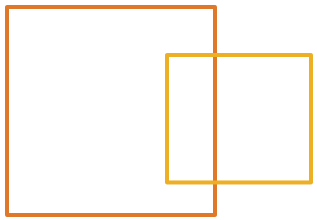
Lab 5a

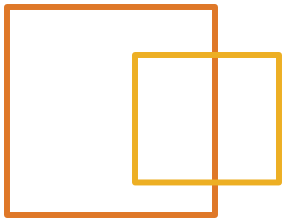




- Inheritance
  - Use the Person class created in Lab 3 (or create it if you did not) and create a subclass called UniversityPerson. Resolve the constructor requirements, and add a new field that indicates if the Person has a teaching role or not. Modify the behavior of the method that returns the prefix so that it returns "Professor" if the person has a teaching role, or "Dr" if not.
  - In your “application” class create an array of six Person references. Initialize the array with three Person objects, and three UniversityPerson objects. Print out the formal addresses for each.
  - Solution: SimpleInheritanceLab

Lab 5b





- Interfaces
  - Define an interface Photographer, that declares a method takePhoto(). Create two classes that implement this interface. One will be called Artist, and this class will be a subclass of your existing Person class. The other class will be SpySatellite, and this has no special parent class. Provide implementations of the takePhoto method in each case so that the method prints out some message that is representative of the nature of the photographer (perhaps the Artist says "say cheese" and the satellite prints "I can read your license plate".
  - Create a class that contains a main method. In the main method, create an array of Photographer, and use this to refer to an Artist and a Satellite. Iterate over the array and ask each element of the array to take a photograph.
  - Solution: PhotoInterfaceLab