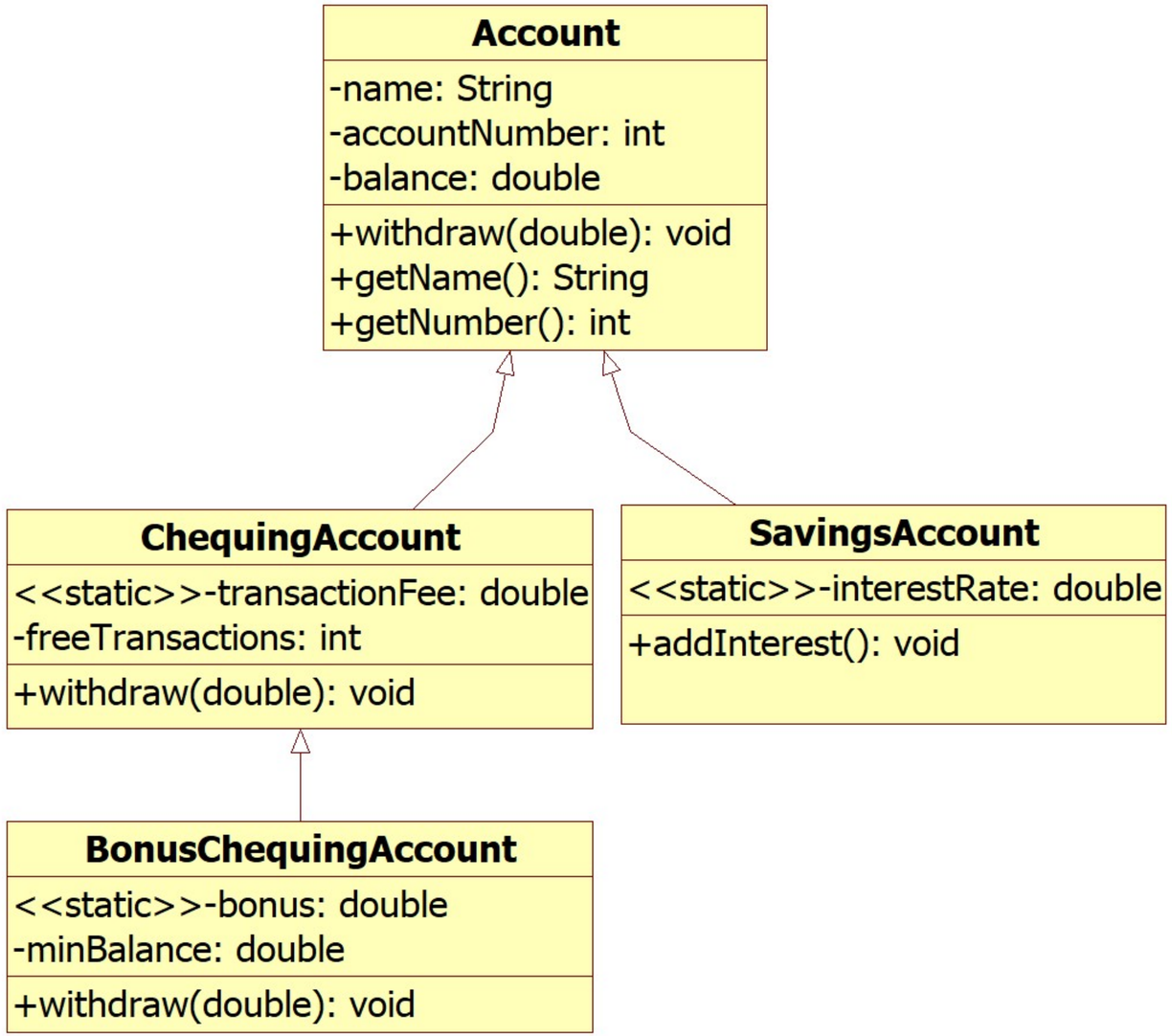
Programming Using Java

Session 5: Class Behaviour

Account Example

- chequing account charges a fee after “x” amount of transactions

- bonus chequing account “pays back” for each transaction



```
public class Account {
  private String name;
  private int accountNumber;
  private double balance;
  …

  public void withdraw(double w) {
     …
  }
  …
}
```

```
public class ChequingAccount extends Account {
  private static double transactionFee = 1.5;
  private int freeTranactions;

  …

  @Override
  public void withdraw(double w) {
    …
  }
}
```

```
public class BankApp {

  public static void main(String[] args) {

    // create object of Account type
    Account a = new Account("jane", 7000);

    // create object of ChequingAccount type
    ChequingAccount ca =
        new ChequingAccount("ed", 7001);

    // is this a valid statement?

    Account a2= new ChequingAccount("jo", 7002);


  }

}
```

Compatible Types

"…a reference can be assigned to a variable of any of its superclasses…"

```java
public class Account {
  private String name;
  private int accountNumber;
  private double balance;
  …

  public void withdraw(double w) {
     …
  }
  …
}
```

```java
public class ChequingAccount extends Account {
  private static double transactionFee = 1.5;
  private int freeTranactions;

  …

  @Override
  public void withdraw(double w) {
    …
  }
}
```

```java
public class BankApp {

  public static void main(String[] args) {

    // create object of Account type
    Account a = new Account("jane", 7000);

    // create object of ChequingAccount type
    ChequingAccount ca =
        new ChequingAccount("ed", 7001);

    Account a2= new ChequingAccount("jo", 7002);

    a.withdraw(10);
    ca.withdraw(10);

    a2.withdraw(10);


  }

}
```

Method Resolution

"…methods are resolved by the reference's dynamic type…"

```
public abstract class Account {
  private String name;
  private int accountNumber;
  private double balance;
  …

  public void withdraw(double w) {
     …
  }
  …
}
```

```
public class ChequingAccount extends Account {
  private static double transactionFee = 1.5;
  private int freeTranactions;

  …

  @Override
  public void withdraw(double w) {
    …
  }
}
```

```
public class BankApp {

  public static void main(String[] args) {

    // create list of various
    // types of accounts
   Account[] aList = new Account[3]
   aList[0] = new SavingsAccount(“jane”, 7000);
   aList[1] = new ChequingAccount(“ed”, 7001);
   aList[2] = new BonusChequingAccount(“jo”, 7002);

    for (Account i : a)
       i.withdraw(50);
  }
}
```

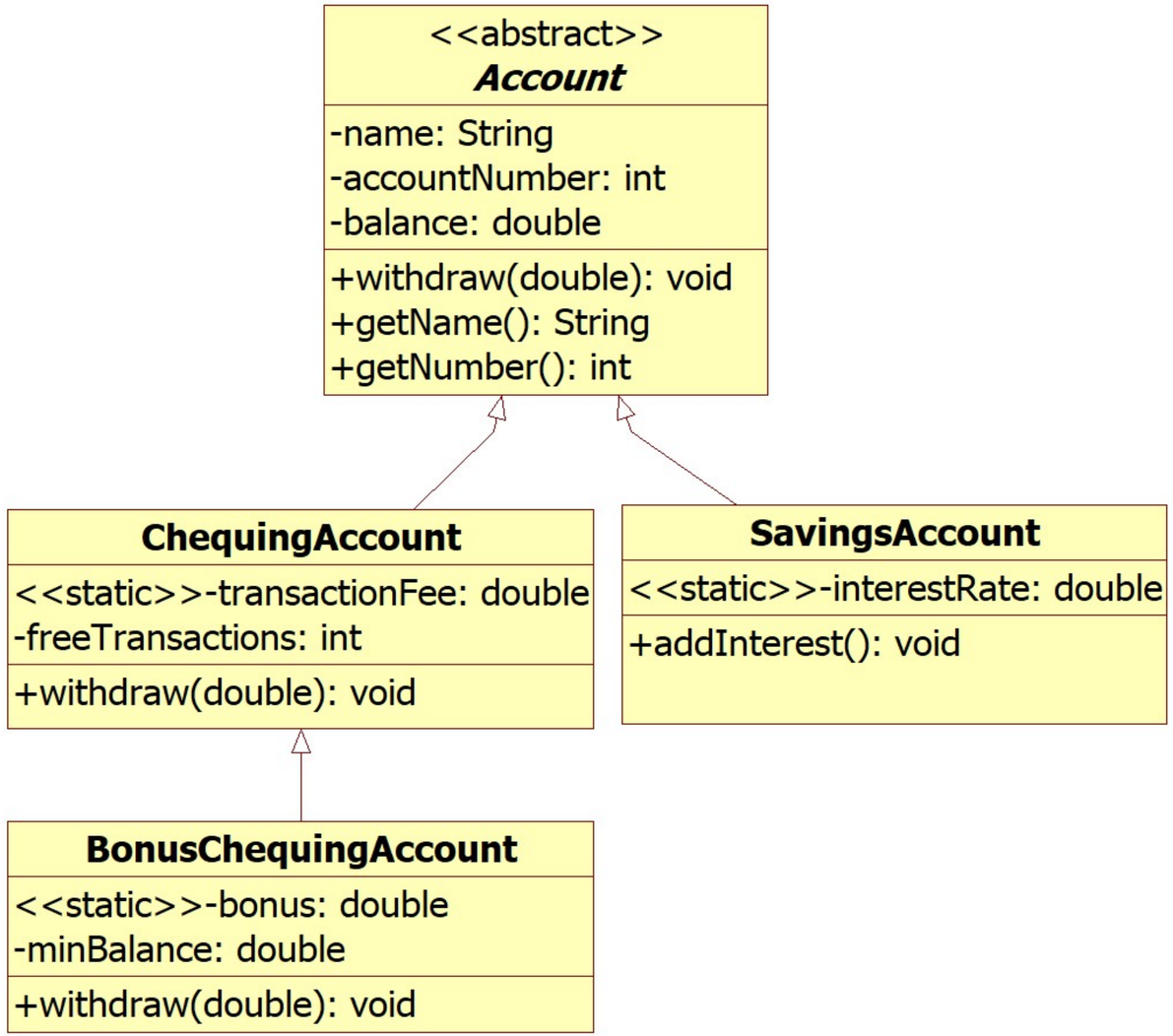
Polymorphic Behaviour

“…method call acts differently depending on context…”

Advanced Class Design

Abstract Classes

- cannot create objects of this type
- ensures structure of subclasses
- methods may be abstract



```
public abstract class Account {
  private String name;
  private int accountNumber;
  private double balance;
  …

  public void withdraw(double w) { this.accountBalance -= w;}
  public void deposit(double w)  { this.accountBalance += w;}
  public String getBalance ()    { return ("$" + this.accountBalance); }
  …
}
```

method signatures

method implementations

```
public class ChequingAccount extends Account {
  …
  @Override
  public void withdraw(double w) { … }
}
```

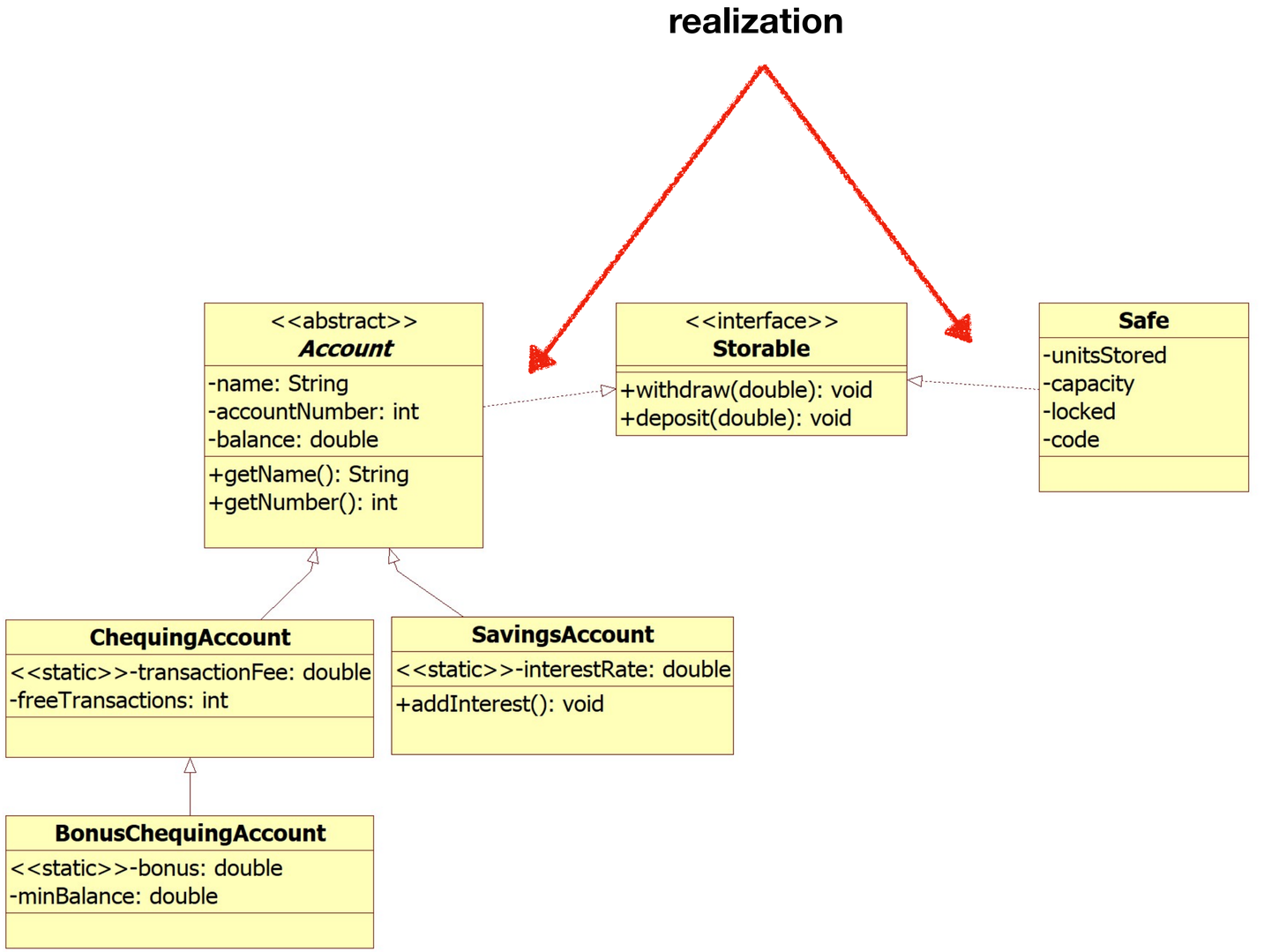
```
public class BankApp {
  public static void main(String[] args) {
    ChequingAccount ca = new ChequingAccount("ed", 7001);
    ca.withdraw(10);
  }}
```

Method Signatures

"…define number and types of arguments required to use a service…"

Interfaces

- are not runtime entites
- are realized by classes
- realized classes need not be related in any way



```
public interface Storable {
  public void withdraw(double w);
  public void deposit(double w);
}
```

```
public class Account implements Storable {
  private String name;
  private int accountNumber;
  private double balance;
  …
  public void withdraw(double w) { … }
  public void deposit(double w) { … }

  …
}
```

```
public class Safe implements Storable {
  private int lock;
  private double size;
  …
  public void withdraw(double w) { … }
  public void deposit(double w) { … }
  …
}
```

```
public class BankApp {

  public static void main(String[] args) {

    // create list of various
    // types of accounts
   Storable[] sList = new Storable[3]
   sList[0] = new Account("jane", 7000);
   sList[1] = new Safe();
   sList[2] = new BonusChequingAccount("jo", 7002);

    for (Storable i : a)
       i.withdraw(50);
  }

}
```

Interfaces

"…defines behaviour that must be defined by a class…"

Source	Rules
superclass (parent)	have the option of inheriting implementation directly…or overriding it to provide custom behaviour. Parent implementation can be accessed via the super keyword. If method was defined as abstract, no implementation exists, and one must be provided
interface	Like an abstract method, an implementation must be provided. super makes no sense here as no parent is being inherited from.
locally defined	implementation must be given in class (unless defined as abstract). If a implementation is given, child class can inherit it.

Class Method Origin Summary

"…take operands and return a boolean result…"