

Section #1: Specification and Documentation

CSE 331 – Spring 2010



Parts of a Specification

- ▶ Usually in-code for Java (Javadoc)
- ▶ Published to clients who use your implementation
- ▶ Two main parts
 - Class Javadoc
 - Method Javadoc

Class Javadoc

- ▶ Class overview – description in English
 - What the class represents
 - Why someone might use it
- ▶ **@specfield** tags – the “parts” or “components” of the abstract object
 - @specfield <name> : <type> // <description>
 - Types independent of actual Java types; e.g. string, integer, sequence, decimal

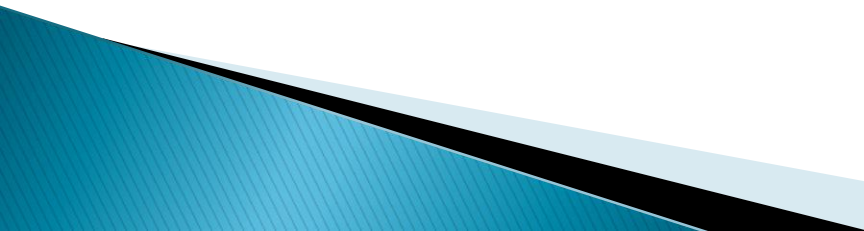
Specfields – Examples

```
/**  
 * <Class overview goes here>  
 * @specfield name: string //name of account owner  
 * @specfield balance: integer //balance of account, in  
   US cents  
 * @specfield transactions: sequence //history of  
   transactions,  
 * //most recent listed first  
 */  
public interface BankAccount {  
    //...
```

Specfields – Exercise

- ▶ See Chain.java

Method Javadoc

- ▶ **@requires** – what is assumed when the method is called
 - ▶ **@modifies** – a *list* of “specfields” identifying what might be modified by the method
 - ▶ **@effects** – how the items in the “modifies” list are affected
 - ▶ **@return** – what the method returns
 - ▶ **@throws** – each of these lists an exception and the conditions under which it will be thrown
 - ▶ Optional description of what the method does in English
- 

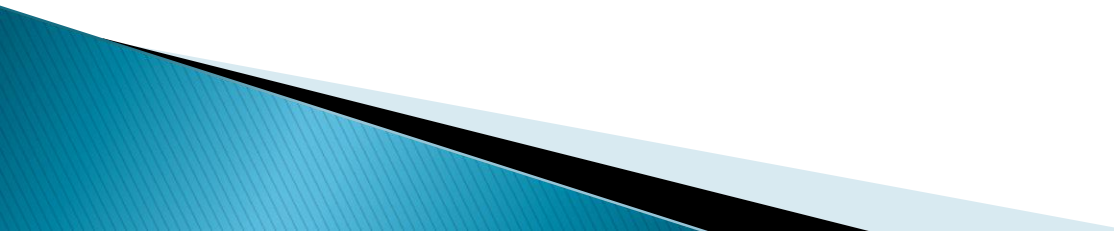
Method Javadoc – Exercise

- ▶ See Chain.java

Class Documentation

- ▶ Not part of the specification, so use regular comment blocks
- ▶ Primarily to help other developers understand how your code works
- ▶ Two main sections we advocate in CSE331
 - Abstraction functions
 - Representation invariants

Abstraction Function

- ▶ Explains the link between the concrete implementation and specification of specfields
 - ▶ Defines the specfields in terms of the actual class fields
 - ▶ Usually mathematical or formal in nature
- 

Abstraction Function – Examples

From PS1, RatNum (rational number):

```
private final int numer;
```

```
private final int denom;
```

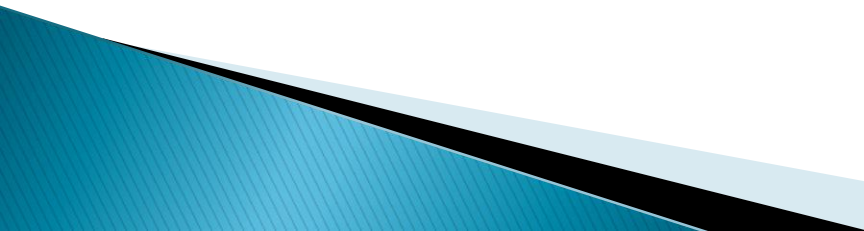
```
// Abstraction Function:
```

```
// A RatNum r is NaN if r.denom = 0,
```

```
// or (r.numer / r.denom) otherwise.
```

▶ See Chain.java

Representation Invariant (RI)

- ▶ Tells what configurations of your class variables are “legal”
 - ▶ Should be true of all instances of your object at all times (otherwise there is a bug)
 - ▶ We will often require you to create a `checkRep()` method for each class that checks the RI for any given instance
 - ▶ You will call `checkRep()` invariant at the end of each public method, at least during testing
- 

Representation Invariant – Examples

- ▶ From RatNum in PS1, the representation invariant is
 - $r.\text{denom} \geq 0 \ \&\& \ (r.\text{denom} > 0 \rightarrow \text{there does not exist integer } i > 1 \text{ such that } r.\text{numer} \bmod i == 0 \text{ and } r.\text{denom} \bmod i == 0)$
 - i.e. the rational number must have a non-negative denominator and be in lowest terms
- ▶ See Chain.java