
CSE 331

Software Design & Implementation

Autumn 2022

Section 10: Review

Administrivia

- HW9 due tomorrow, Friday 12/9
 - Implement feedback from HW7!
 - Document your React code and Spark Server!
- Final on Tuesday, December 13 at 2:30
 - Review session on Monday (December 12) from 11 AM – 1 PM in CSE2 G10. Come with questions!
- Any questions?

Agenda

- HW9 demos
- Review
 - Reasoning, Specifications, ADTs (RI & AF), Testing, Defensive Programming, Equals and Hash Code, Exceptions, Subtyping, Generics
- Design Patterns

HW9 Demos

- Anyone want to show off their HW9?

Reasoning with `if` (reminders)

```
{{ x < 0 }}
```

↓

```
if (y > x)
```

```
    {{ ? }}
```

```
    y = x;
```

```
    {{ ? }}
```

```
else
```

```
    {{ ? }}
```

```
    y = y - 1;
```

```
    {{ ? }}
```

↑

```
{{ x < 0 & y < 0 }}
```

This does not mean you should backwards reason through the **else** branch and forward reason through the **if** branch

Reasoning with `if` (reminders)

```
{ { x < 0 } }  
↓  
if (y > x)  
    { { x < 0 & y > x } }  
    y = x;  
    { { ? } }  
else  
    { { x < 0 & y <= x } }  
    y = y - 1;  
    { { ? } }  
↑  
{ { x < 0 & y < 0 } }
```

This does not mean you should backwards reason through the **else** branch and forward reason through the **if** branch

Forward reason into the precondition of both branches

Reasoning with `if` (reminders)

```
{ { x < 0 } }  
↓  
if (y > x)  
    { { x < 0 & y > x } }  
    y = x;  
    { { x < 0 & y < 0 } }  
else  
    { { x < 0 & y <= x } }  
    y = y - 1;  
    { { x < 0 & y < 0 } }  
↑  
{ { x < 0 & y < 0 } }
```

This does not mean you should backwards reason through the **else** branch and forward reason through the **if** branch

Forward reason into the precondition of both branches

Backwards reason into the postcondition of both branches

Reasoning with `if` (reminders)

`{{ x < 0 }}`

↓

`if (y > x)`

`{{ x < 0 & y > x }}`

`y = x;`

`{{ x < 0 & y < 0 }}`

`else`

`{{ x < 0 & y <= x }}`

`y = y - 1;`

`{{ x < 0 & y < 0 }}`

↑

`{{ x < 0 & y < 0 }}`

Is the code correct?

Yep!

Backwards Reasoning Reminders

What goes here?

```
    {{ ? }}  
↑  k = k + 1;  
    {{ k = 2 }}
```

Backwards Reasoning Reminders

What goes here?

X `{ { k = k + 1 } }`

↑ `k = k + 1;`
`{ { k = 2 } }`

Remember: An assertion is a true / false claim (proposition) about the **state** at a given point during execution

Assertions like `k = k + 1` make no sense because it is a contradiction and will always evaluate to **false**!

Try putting `assert (k == k + 1) ;` in your code!

Backwards Reasoning Reminders

What goes here?

✓ $\{\{ k + 1 = 2 \}\} \Leftrightarrow \{\{ k = 1 \}\}$
↑ `k = k + 1;` ← Note: single equals
 $\{\{ k = 2 \}\}$

In backwards reasoning, we take the **bottom** assertion and **replace** all instances of the **left-hand** side of the assignment statement with the **right-hand** side of the assignment statement!

Do **NOT** write `k = 2 & k = k + 1`

- `assert(k == 2 && k == k + 1);` will always fail (contradiction!)

Backwards Reasoning Reminders

What goes here?

```
{ { inv: sum = A[0] + ... + A[i-1] } }  
while (i != A.length) {  
    sum = sum + A[i];  
    i = i + 1;  
    { { ? } }  
↑  
}
```

Backwards Reasoning Reminders

What goes here?

```
{ { inv: sum = A[0] + ... + A[i-1] } }  
while (i != A.length) {  
    sum = sum + A[i];  
    i = i + 1;  
    X{ { inv  $\wedge$  i  $\neq$  n } }  
    ↑  
}
```

When backwards reasoning at the bottom of the loop, we don't know if the loop condition still holds. We only know that the invariant holds. By including that the loop condition holds at the bottom of the loop, you're indicating we never exit the loop!

Backwards Reasoning Reminders

What goes here?

```
{ { inv: sum = A[0] + ... + A[i-1] } }  
while (i != A.length) {  
    sum = sum + A[i];  
    i = i + 1;  
    ✓ { { inv } }  
↑  
}
```

Reasoning w/ Loops (again!)

We will fill in the implementation for `runLengthEncode`:

```
int runLengthEncode(String str, int n, char[] chars, int[] lens)
```

- We need to *encode* the first `n` characters of `str`, writing into `chars` and `lens`.
- Returns `t` such that

$$\text{str}[0\dots n-1] = \text{chars}[0] * \text{lens}[0] + \dots + \text{chars}[t-1] * \text{lens}[t-1]$$

- Ex: after encoding "aaabbccccaadd" with `n = 14`:
 - `chars[0...4] = ['a', 'b', 'c', 'a', 'd']`
 - `lens[0...4] = [3 , 2 , 4 , 2 , 3]`
 - `returns 5`

Reasoning w/ Loops (again!)

- Ex: after encoding "aaabbccccaadd" with $n = 14$:
 - `chars[0..4] = ['a', 'b', 'c', 'a', 'd']`
 - `lens[0..4] = [3 , 2 , 4 , 2 , 3]`
 - `returns 5`
- Returns t such that
$$\text{str}[0..n-1] = \text{chars}[0] * \text{lens}[0] + \dots + \text{chars}[t-1] * \text{lens}[t-1]$$
 - In other words, return $t = \#$ of chars that we overwrote in `chars`
= $\#$ of ints that we overwrote in `lens`
- Note: can't have two of the same char in a row in `chars`!
- All of `lens[0..t-1]` must be greater than 0!
- You can assume that `str` and both arrays have lengths greater than or equal to n and that `str` does not contain `'\0'`.

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
    {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
    int i =    ;  
    int j =    ;  
    char cur =    ;  
    {{ inv: pre and  
        str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
        (i = 0 or cur = str[i-1]) }}  
    while (_____) {  
        ...  
    }  
  
    {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
    return _____;  
}
```

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
    {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
    int i = 0;  
    int j = -1;  
    char cur = '\0';  
    {{ inv: pre and  
        str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
        (i = 0 or cur = str[i-1]) }}  
    while (_____) {  
        ...  
    }  
  
    {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
    return _____;  
}
```

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
    {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
    int i = 0;  
    int j = -1;           Does our invariant hold with this i, j, and curr?  
    char cur = '\0';  
    {{ inv: pre and  
        str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
        (i = 0 or cur = str[i-1]) }}  
    while (_____) {  
        ...  
    }  
  
    {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
    return _____;  
}
```

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
    {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
    int i = 0;  
    int j = -1; pre holds because all the related  
variables were untouched  
    char cur = '\0';  
    {{ inv: pre and  
        str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
        (i = 0 or cur = str[i-1]) }}  
    while (_____) {  
        ...  
    }  
  
    {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
    return _____;  
}
```

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
    {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
    int i = 0;  
    int j = -1;  
    char cur = '\0';  
    {{ inv: pre and  
        str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
        (i = 0 or cur = str[i-1]) }}  
    while (_____) {  
        ...  
    }  
  
    {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
    return _____;  
}
```

The next part holds since there are no indexes k such that $0 \leq k \leq i-1 = -1$. There are also no such indexes m such that $0 \leq m \leq j = -1$. This is vacuously true.

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
    {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
    int i = 0;  
    int j = -1;  
    char cur = '\0';  
    {{ inv: pre and  
        str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
        (i = 0 or cur = str[i-1]) }}  
    while (_____) {  
        ...  
    }  
  
    {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
    return _____;  
}
```

The next two parts hold for the same reasoning as previous. This statement is vacuously true since there are no indexes m such that $0 \leq m \leq j = -1$

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
    {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
    int i = 0; Finally, this last part holds because we have i = 0.  
    int j = -1;  
    char cur = '\0';  
    {{ inv: pre and  
        str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
        (i = 0 or cur = str[i-1]) }}  
    while (_____) {  
        ...  
    }  
  
    {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
    return _____;  
}
```


Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
  {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
  int i = 0;  
  int j = -1;  
  char cur = '\0';  
  {{ inv: pre and  
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
    (i = 0 or cur = str[i-1]) }}  
  while (_____) {  
    ... How can we make the postcondition hold after the loop?  
  }  
  
  {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
  return _____;  
}
```

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
  {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
  int i = 0;  
  int j = -1;  
  char cur = '\0';  
  {{ inv: pre and  
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
    (i = 0 or cur = str[i-1]) }}  
  while (i != n) {  
    ...  
  }  
  
  {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
  return _____;  
}
```

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
    {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
    int i = 0;  
    int j = -1;  
    char cur = '\0';  
    {{ inv: pre and  
        str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
        (i = 0 or cur = str[i-1]) }}  
    while (i != n) {  
        ...  
    }  
    {{ inv and i = n }}  
    {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
    return _____;  
}
```

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
  {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
  int i = 0;  
  int j = -1;  
  char cur = '\0';  
  {{ inv: pre and  
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
    (i = 0 or cur = str[i-1]) }}  
  while (i != n) {  
    ...  
  }  
  {{ inv and i = n }}  
  {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
  return _____;  
}
```

These two match if we substitute $i = n$.

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
  {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
  int i = 0;  
  int j = -1;  
  char cur = '\0';  
  {{ inv: pre and  
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
    (i = 0 or cur = str[i-1]) }}  
  while (i != n) {  
    ...  
  }  
  {{ inv and i = n }}  
  {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
  return _____;  
}
```

These two match exactly.

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
  {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
  int i = 0;  
  int j = -1;  
  char cur = '\0';  
  {{ inv: pre and  
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
    (i = 0 or cur = str[i-1]) }}  
  while (i != n) {  
    ...  
  }  
  {{ inv and i = n }}  
  {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
  return _____;  
}
```

The assertion above has additional facts, making it stronger.
Our postcondition is weaker since it excludes those facts!
A stronger assertion implies a weaker one!

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
  {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
  int i = 0;  
  int j = -1;  
  char cur = '\0';  
  {{ inv: pre and  
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
    (i = 0 or cur = str[i-1]) }}  
  while (i != n) {  
    ...  
  }  
  {{ inv and i = n }}  
  {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
  return _____;  
}
```

What should we return? Recall:

Returns t such that

$str[0...n-1] = chars[0] * lens[0] + \dots + chars[t-1] * lens[t-1]$

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
  {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
  int i = 0;  
  int j = -1;  
  char cur = '\0';  
  {{ inv: pre and  
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
    (i = 0 or cur = str[i-1]) }}  
  while (i != n) {  
    ...  
  }  
  {{ inv and i = n }}  
  {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j]  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
  return _____;  
}
```

What should we return? Recall:

Returns t such that

$str[0...n-1] = chars[0] * lens[0] + \dots + chars[t-1] * lens[t-1]$

Reasoning w/ Loops (again!)

```
int runLengthEncode(String str, int n, char[] chars, int[] lens) {  
    {{ pre: 0 < n <= str.length, chars.length, lens.length }}  
    int i = 0;  
    int j = -1;  
    char cur = '\0';  
    {{ inv: pre and  
        str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
        (i = 0 or cur = str[i-1]) }}  
    while (i != n) {  
        ...  
    }  
    {{ inv and i = n }}  
    {{ post: str[0...n-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
        chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
        lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 }}  
    return j + 1;  
}
```

Since we know that $j = t - 1$, we can solve for t !
We get $j + 1 = t$.

Reasoning w/ Loops (again!)

```
{ { inv: pre and  
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
    (i = 0 or cur = str[i-1]) } }
```

```
while (i != n) {
```

We have examined our string through index $i - 1$

What do we need to do next?

```
}
```

Reasoning w/ Loops (again!)

```
{ { inv: pre and
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and
    (i = 0 or cur = str[i-1]) } }
```

```
while (i != n) {
```

We need to examine the character at index i

How do we process a character?

```
}
```

Reasoning w/ Loops (again!)

```
{ { inv: pre and  
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and  
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and  
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and  
    (i = 0 or cur = str[i-1]) } }
```

```
while (i != n) {
```

We need to examine the character at index i

How do we process a character?

There are two cases: this character needs a new spot in `chars` and `lens` or it belongs at index j

How do we differentiate these two?

```
}
```

Reasoning w/ Loops (again!)

```
{ { inv: pre and
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and
    (i = 0 or cur = str[i-1]) } }
while (i != n) {
    if (str.charAt(i) == cur) {
        // same as last character, goes into index j
    } else {
        // different from last character, needs new spot
    }
}
}
```

Reasoning w/ Loops (again!)

```
{ { inv: pre and
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and
    (i = 0 or cur = str[i-1]) } }
while (i != n) {
    if (str.charAt(i) == cur) {
        // same as last character, goes into index j
        What goes here?
    } else {
        // different from last character, needs new spot

    }
}
```

Reasoning w/ Loops (again!)

```
{ { inv: pre and
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and
    (i = 0 or cur = str[i-1]) } }
while (i != n) {
    if (str.charAt(i) == cur) {
        // same as last character, goes into index j
        lens[j] = lens[j] + 1;
    } else {
        // different from last character, needs new spot

    }
}
```

Reasoning w/ Loops (again!)

```
{ { inv: pre and
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and
    (i = 0 or cur = str[i-1]) } }
while (i != n) {
    if (str.charAt(i) == cur) {
        // same as last character, goes into index j
        lens[j] = lens[j] + 1;
    } else {
        // different from last character, needs new spot
        What goes here?

    }
}
```


Reasoning w/ Loops (again!)

```
{ { inv: pre and
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and
    (i = 0 or cur = str[i-1]) } }
while (i != n) {
    if (str.charAt(i) == cur) {
        // same as last character, goes into index j
        lens[j] = lens[j] + 1;
    } else {
        // different from last character, needs new spot
        j = j + 1;
        cur = str.charAt(i);
        chars[j] = cur;
        lens[j] = 1;
    }
}
```

Reasoning w/ Loops (again!)

```
{ { inv: pre and
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and
    (i = 0 or cur = str[i-1]) } }
while (i != n) {
    if (str.charAt(i) == cur) {
        // same as last character, goes into index j
        lens[j] = lens[j] + 1;
    } else {
        // different from last character, needs new spot
        j = j + 1;
        cur = str.charAt(i);
        chars[j] = cur;
        lens[j] = 1;
    }
}
```

What's left?

```
}
```

Reasoning w/ Loops (again!)

```
{ { inv: pre and
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and
    (i = 0 or cur = str[i-1]) } }
while (i != n) {
    if (str.charAt(i) == cur) {
        // same as last character, goes into index j
        lens[j] = lens[j] + 1;
    } else {
        // different from last character, needs new spot
        j = j + 1;
        cur = str.charAt(i);
        chars[j] = cur;
        lens[j] = 1;
    }
    i = i + 1;
}
```

Reasoning w/ Loops (again!)

```
{ { inv: pre and
    str[0...i-1] = chars[0] * lens[0] + ... + chars[j] * lens[j] and
    chars[0] != chars[1], ..., chars[j-1] != chars[j] and
    lens[0] > 0, lens[1] > 0, ..., lens[j] > 0 and
    (i = 0 or cur = str[i-1]) } }
while (i != n) {
    if (str.charAt(i) == cur) {
        // same as last character, goes into index j
        lens[j] = lens[j] + 1;
    } else {
        // different from last character, needs new spot
        j = j + 1;
        cur = str.charAt(i);
        chars[j] = cur;
        lens[j] = 1;
    }
    i = i + 1;
}
```

Only the invariant must hold at the end of each loop iteration.
Does it hold?

Testing

What would be some good test cases for this method?

Note: @param tags omitted.

```
/** A very mysterious method with a great description
 *  @requires x > 0 and y > 0
 *  @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
    if (x == y) { throw new IllegalArgumentException(); }
    return (x > y) ? (x + y) : (x - y);
}
```

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 * @requires x > 0 and y > 0
 * @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
    if (x == y) { throw new IllegalArgumentException(); }
    return (x > y) ? (x + y) : (x - y);
}
```

Is `mystery(0, 0)` a good test case?

No. Its behavior is undefined. We cannot test for undefined behavior!

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 * @requires x > 0 and y > 0
 * @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
    if (x == y) { throw new IllegalArgumentException(); }
    return (x > y) ? (x + y) : (x - y);
}
```

Is `mystery(1, 1)` a good test case?

Yes – we are testing for an `IllegalArgumentException` being thrown.

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 * @requires x > 0 and y > 0
 * @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
    if (x == y) { throw new IllegalArgumentException(); }
    return (x > y) ? (x + y) : (x - y);
}
```

Is `mystery(2, 2)` a good test case?

We already tested for this with `mystery(1, 1)`. This is not testing anything new.

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 * @requires x > 0 and y > 0
 * @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
    if (x == y) { throw new IllegalArgumentException(); }
    return (x > y) ? (x + y) : (x - y);
}
```

Is `mystery(3, 2)` a good test case?

Yes!

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 * @requires x > 0 and y > 0
 * @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
    if (x == y) { throw new IllegalArgumentException(); }
    return (x > y) ? (x + y) : (x - y);
}
```

Is `mystery(4, 10)` a good test case?

Yes!

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 * @requires x > 0 and y > 0
 * @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
    if (x == y) { throw new IllegalArgumentException(); }
    return (x > y) ? (x + y) : (x - y);
}
```

Is `mystery(400000, 1000000)` a good test case?

It's not testing anything new.

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 * @requires x > 0 and y > 0
 * @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
    if (x == y) { throw new IllegalArgumentException(); }
    return (x > y) ? (x + y) : (x - y);
}
```

Is `mystery(42, -42)` a good test case?

No! It's testing for undefined behavior!

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 * @requires x > 0 and y > 0
 * @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
    if (x == y) { throw new IllegalArgumentException(); }
    return (x > y) ? (x + y) : (x - y);
}
```

Is `mystery(null, null)` a good test case?

No! `int` is a primitive, and it cannot be null.

Arrays and Covariance

- In Java, arrays are **covariant**, meaning:
 - Array of type **E[]** can hold elements **E** and subtypes of **E**.
 - An array of type **E1** is a subtype of **E[]** if **E1** is a subtype of **E**.

```
Number[] nums = new Integer[10]; ✓
```

```
Integer[] ints = new Number[10]; ✗
```

```
Integer[] ints = new int[10]; ✗
```

```
Number[] nums = new int[10]; ✗
```

Which of these are legal?

```
Number[] nums = new Number[10];
Number[] numints = new Integer[10];
Integer[] ints = new Integer[10];
public void mystery1(Integer[] ints) { }
public void mystery2(Number[] nums) { }
```

```
nums[0] = 1.23;
```

```
numints[0] = 1.23;
```

```
mystery1(nums);
```

```
mystery2(ints);
```

Is `Integer[]` a true subtype of `Number[]`?

Which of these are legal?

```
Number[] nums = new Number[10];  
Number[] numints = new Integer[10];  
Integer[] ints = new Integer[10];  
public void mystery1(Integer[] ints) { }  
public void mystery2(Number[] nums) { }
```

nums[0] = 1.23; ✓

numints[0] = 1.23; ✗

mystery1(nums); ✗

mystery2(ints); ✓

Is Integer[] a true subtype of Number[]? ✗

Subtypes & Subclasses

- Subtypes are substitutable for supertypes
- If **Foo** is a subtype of **Bar**,
 G<Foo> is a **NOT** a subtype of **G<Bar>**
- Aliasing resulting from this would let you add objects of type **Bar** to **G<Foo>**, which would be bad!
- Example:

```
List<String> ls = new ArrayList<String>();  
List<Object> lo = ls;  
lo.add(new Object());  
String s = ls.get(0);
```

- Subclassing is done to reuse code (extends)
- A subclass can override methods in its superclass

Typing and Generics

- `<?>` is a wildcard for unknown
 - Lower bounded wildcard `<? super SomeClass>` (superclass)
 - Can only insert items with type `SomeClass`, or a type that extends `SomeClass`
 - Why? Because we can cast that object into `SomeClass`.
 - Illegal to retrieve as type other than `Object`.
- What types can you put here?
 - `List<? super Number> lsn = new ArrayList<_____>();`
 - `Object`
 - `Number`
 - `Number` *is* an `Object`. But an `Object` might not be a `Number`.

Typing and Generics

- What types can you put here?
 - **List<? super Number> lsn = new ArrayList<_____>();**
 - Object
 - Number
 - Number *is* an Object. But an Object might not be a Number.
- Object o = new Object(); lsn.add(o); ❌
- Number n = 4; lsn.add(n); ✓
- Integer i = 5; lsn.add(i); ✓

Remember: ArrayList<Number> must hold Numbers. We cannot add `o` because an Object cannot be cast into a Number. But we can add `i` because an Integer can be cast into a Number.

Typing and Generics

- What types can you put here?
 - **List<? super Number> lsn = new ArrayList<_____>();**
 - Object
 - Number
 - Number *is* an Object. But an Object might not be a Number.
 - Object o = lsn.get(0); ✓
 - Number n = lsn.get(0); ✗
 - Integer i = lsn.get(0); ✗

Since `lsn` *could* be an `ArrayList<Object>`, we can only pull Objects out of here.

Typing and Generics

- `<?>` is a wildcard for unknown
 - Upper bounded wildcard: `<? extends _____>` (subclass)
 - Safe to read from: result will be the type after **extends**
 - Illegal to write into (no calls to add!) because we can't guarantee type safety.
 - What types can you put here?
 - **List<? extends Number> lei = new ArrayList<_____>();**
 - Number
 - Integer, Float, Double, Long...
 - Some other class that extends Integer.
 - Some other class that extends the class that extended Integer... (infinitely downwards)
 - Anything that extends Number will still be a Number. But, Number might not be an Integer, or any of its subclasses.

Typing and Generics

- What types can you put here?
 - **List<? extends Number> lei = new ArrayList<_____>();**
 - Number
 - Integer, Float, Double, Long...
 - Some other class that extends Integer.
 - Some other class that extends the class that extended Integer... (infinitely downwards)
 - Anything that extends Number will still be a Number. But, Number might not be an Integer, or any of its subclasses.
 - Object o = new Object(); len.add(o); ❌
 - Number n = 4; len.add(n); ❌
 - Integer i = 5; len.add(i); ❌

We cannot add anything because there is no lower bound on the actual type of the List.

Typing and Generics

- What types can you put here?
 - **List<? extends Number> lei = new ArrayList<_____>();**
 - Number
 - Integer, Float, Double, Long...
 - Some other class that extends Integer.
 - Some other class that extends the class that extended Integer... (infinitely downwards)
 - Anything that extends Number will still be a Number. But, Number might not be an Integer, or any of its subclasses.
 - Object o = len.get(0); ✓
 - Number n = len.get(0); ✓
 - Integer i = len.get(0); ✗

When we retrieve an element, it must be of type Number. A Number is an Object, but a Number might not be an Integer.

Subtypes & Subclasses

Given the below classes which one of the statements in the box are legal?

```
class Student extends Object { ... }
```

```
class CSEStudent extends Student { ... }
```

```
List<Student> ls;
```

```
List<? extends Student> les;
```

```
List<? super Student> lss;
```

```
List<CSEStudent> lcse;
```

```
List<? extends CSEStudent> lecse;
```

```
List<? super CSEStudent> lscse;
```

```
Student scholar;
```

```
CSEStudent hacker;
```

```
ls = lcse;
```

```
les = lscse;
```

```
lcse = lscse;
```

```
les.add(scholar);
```

```
lscse.add(scholar);
```

```
lss.add(hacker);
```

```
scholar = lscse.get(0);
```

```
hacker = lecse.get(0);
```


Subtypes & Subclasses

```
class Student extends Object { ... }  
class CSEStudent extends Student { ... }
```

```
List<Student> ls;  
List<? extends Student> les;  
List<? super Student> lss;  
List<CSEStudent> lcse;  
List<? extends CSEStudent> lecse;  
List<? super CSEStudent> lscse;  
Student scholar;  
CSEStudent hacker;
```

```
ls = lcse;      X  
les = lscse;  
lcse = lscse;  
les.add(scholar);  
lscse.add(scholar);  
lss.add(hacker);  
scholar = lscse.get(0);  
hacker = lecse.get(0);
```

Subtypes & Subclasses

```
class Student extends Object { ... }  
class CSEStudent extends Student { ... }
```

```
List<Student> ls;  
List<? extends Student> les;  
List<? super Student> lss;  
List<CSEStudent> lcse;  
List<? extends CSEStudent> lecse;  
List<? super CSEStudent> lscse;  
Student scholar;  
CSEStudent hacker;
```

```
ls = lcse;      X  
les = lscse;   X  
lcse = lscse;  
les.add(scholar);  
lscse.add(scholar);  
lss.add(hacker);  
scholar = lscse.get(0);  
hacker = lecse.get(0);
```

Subtypes & Subclasses

```
class Student extends Object { ... }  
class CSEStudent extends Student { ... }
```

```
List<Student> ls;  
List<? extends Student> les;  
List<? super Student> lss;  
List<CSEStudent> lcse;  
List<? extends CSEStudent> lecse;  
List<? super CSEStudent> lscse;  
Student scholar;  
CSEStudent hacker;
```

```
ls = lcse;      X  
les = lscse;   X  
lcse = lscse;  X  
les.add(scholar);  
lscse.add(scholar);  
lss.add(hacker);  
scholar = lscse.get(0);  
hacker = lecse.get(0);
```

Subtypes & Subclasses

```
class Student extends Object { ... }  
class CSEStudent extends Student { ... }
```

```
List<Student> ls;  
List<? extends Student> les;  
List<? super Student> lss;  
List<CSEStudent> lcse;  
List<? extends CSEStudent> lecse;  
List<? super CSEStudent> lscse;  
Student scholar;  
CSEStudent hacker;
```

```
ls = lcse;      X  
les = lscse;   X  
lcse = lscse;  X  
les.add(scholar); X  
lscse.add(scholar);  
lss.add(hacker);  
scholar = lscse.get(0);  
hacker = lecse.get(0);
```

Subtypes & Subclasses

```
class Student extends Object { ... }  
class CSEStudent extends Student { ... }
```

```
List<Student> ls;  
List<? extends Student> les;  
List<? super Student> lss;  
List<CSEStudent> lcse;  
List<? extends CSEStudent> lecse;  
List<? super CSEStudent> lscse;  
Student scholar;  
CSEStudent hacker;
```

```
ls = lcse;      X  
les = lscse;   X  
lcse = lscse;  X  
les.add(scholar); X  
lscse.add(scholar); X  
lss.add(hacker);  
scholar = lscse.get(0);  
hacker = lecse.get(0);
```

Subtypes & Subclasses

```
class Student extends Object { ... }  
class CSEStudent extends Student { ... }
```

```
List<Student> ls;  
List<? extends Student> les;  
List<? super Student> lss;  
List<CSEStudent> lcse;  
List<? extends CSEStudent> lecse;  
List<? super CSEStudent> lscse;  
Student scholar;  
CSEStudent hacker;
```

```
ls = lcse;      X  
les = lscse;   X  
lcse = lscse;  X  
les.add(scholar); X  
lscse.add(scholar); X  
lss.add(hacker); 😊  
scholar = lscse.get(0);  
hacker = lecse.get(0);
```

Subtypes & Subclasses

```
class Student extends Object { ... }  
class CSEStudent extends Student { ... }
```

```
List<Student> ls;  
List<? extends Student> les;  
List<? super Student> lss;  
List<CSEStudent> lcse;  
List<? extends CSEStudent> lecse;  
List<? super CSEStudent> lscse;  
Student scholar;  
CSEStudent hacker;
```

```
ls = lcse;      X  
les = lscse;   X  
lcse = lscse;  X  
les.add(scholar); X  
lscse.add(scholar); X  
lss.add(hacker); 😊  
scholar = lscse.get(0); X  
hacker = lecse.get(0);
```

Subtypes & Subclasses

```
class Student extends Object { ... }  
class CSEStudent extends Student { ... }
```

```
List<Student> ls;  
List<? extends Student> les;  
List<? super Student> lss;  
List<CSEStudent> lcse;  
List<? extends CSEStudent> lecse;  
List<? super CSEStudent> lscse;  
Student scholar;  
CSEStudent hacker;
```

```
ls = lcse;      X  
les = lscse;   X  
lcse = lscse;  X  
les.add(scholar); X  
lscse.add(scholar); X  
lss.add(hacker); 😊  
scholar = lscse.get(0); X  
hacker = lecse.get(0); 😊
```


equals for a parameterized class

```
class Node<E> {
```

```
...
```

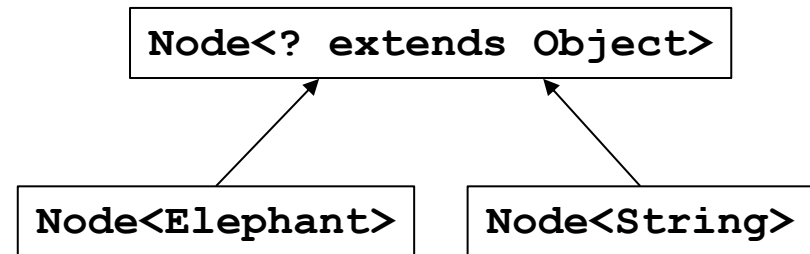
```
@Override
```

```
public boolean equals(Object obj) {  
    if (!(obj instanceof Node<?>)) {  
        return false;  
    }  
    Node<?> n = (Node<?>) obj;  
    return this.data().equals(n.data());  
}
```

Works if the type of obj is Node<Elephant> or Node<String> or ...

```
Node<?> n = (Node<?>) obj;  
return this.data().equals(n.data());
```

Leave it to here to “do the right thing” if **this** and **n** differ on element type



More Generics

```
public class Box<E> { }
```

- **Integer** is a subtype of **Number**.
- Covariant: **Box<Integer>** is a subtype of **Box<Number>**
- Contravariant: **Box<Number>** is a subtype of **Box<Integer>**
- Invariant: neither is a subtype of the other.
 - i.e. **Box<Number>** and **Box<Integer>** are unconvertible types.
- In Java, generics are invariant.

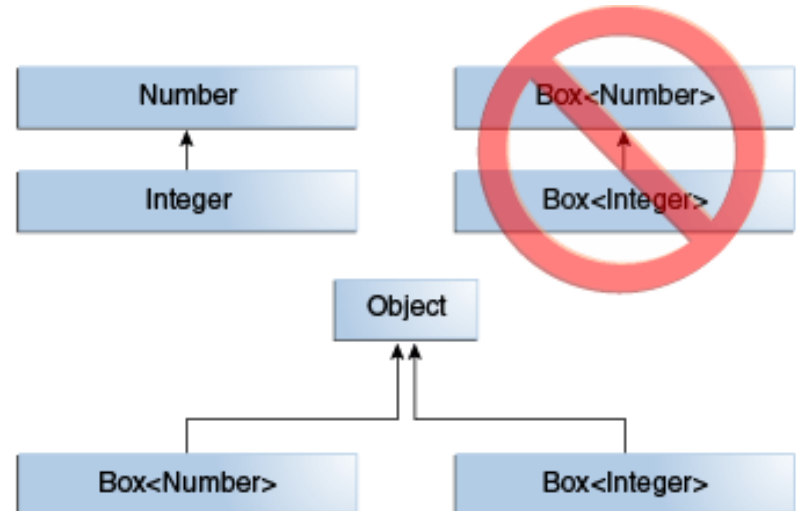
More Generics

```
class Box<E> { }
```

Is this legal?

```
public Box<Number> mystery() {  
    return new Box<Integer>();  
}
```

Nope. `Box<Integer>` is not a subtype of `Box<Number>`, even though `Integer` is a subtype of `Number`.

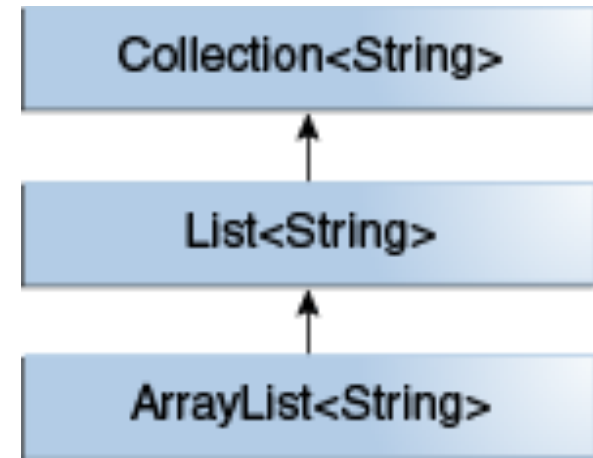


More Generics

Is this legal?

```
public List<String> mystery() {  
    return new ArrayList<String>();  
}
```

Yep. This is fine. As long as the type argument does not change, subtyping is preserved.

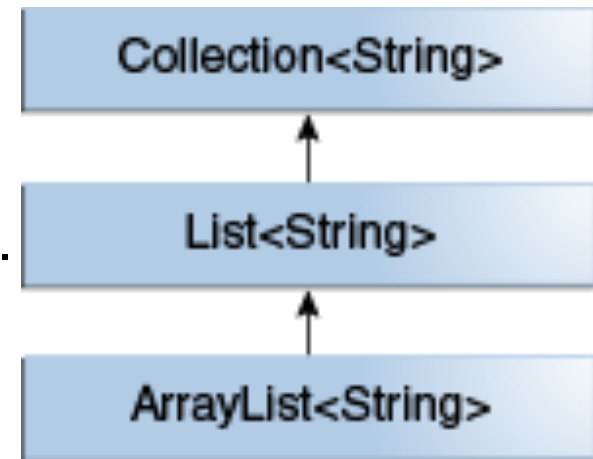


More Generics

Is this legal?

```
public ArrayList<String> mystery() {  
    Collection<String> a = new ArrayList<String>();  
    return a;  
}
```

No. The initialization of `a` is fine, but the return is not. `Collection<String>` is not a subtype of `ArrayList<String>`.



Subclasses & Overriding

```
class Foo extends Object {  
    Shoe m(Shoe x, Shoe y) { ... }  
}
```

```
class Bar extends Foo {...}
```

Overriding a method:

- A method overrides a superclass method only if it has the same name and exact same argument types
 - **Covariant** return types (must be a subtype of the original)

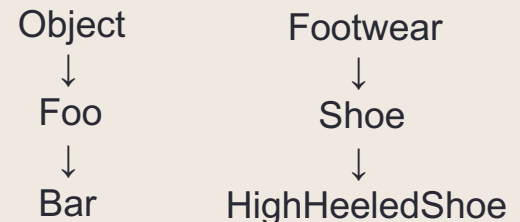
Overloading a method:

- A method overloads a method only if it has the same name and different argument types than another existing method

Method Declarations in Bar

Given the class in the purple box, determine whether the method declarations for the method `Shoe()` inside **Bar** class are overriding or overloading it?

- The result is method overriding
- The result is method overloading
- The result is a type-error
- None of the above

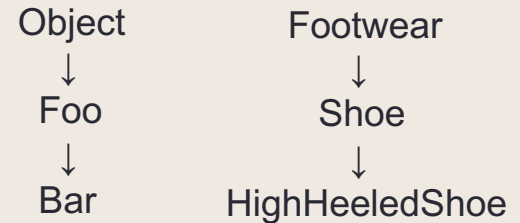


- **FootWear m(Shoe x, Shoe y) { ... }**
- **Shoe m(Shoe q, Shoe z) { ... }**
- **HighHeeledShoe m(Shoe x, Shoe y) { ... }**
- **Shoe m(FootWear x, HighHeeledShoe y) { ... }**
- **Shoe m(FootWear x, FootWear y) { ... }**
- **Shoe m(Shoe x, Shoe y) { ... }**
- **Shoe m(HighHeeledShoe x, HighHeeledShoe y) { ... }**
- **Shoe m(Shoe y) { ... }**
- **Shoe z(Shoe x, Shoe y) { ... }**

```
class Foo extends Object {  
    Shoe m(Shoe x, Shoe y){ ... }  
}  
  
class Bar extends Foo {...}
```

Method Declarations in Bar

- The result is method overriding
- The result is method overloading
- The result is a type-error
- None of the above



- **FootWear m(Shoe x, Shoe y) { ... }** **type-error**
- **Shoe m(Shoe q, Shoe z) { ... }** **overriding**
- **HighHeeledShoe m(Shoe x, Shoe y) { ... }** **overriding**
- **Shoe m(FootWear x, HighHeeledShoe y) { ... }** **overloading**
- **Shoe m(FootWear x, FootWear y) { ... }** **overloading**
- **Shoe m(Shoe x, Shoe y) { ... }** **overriding**
- **Shoe m(HighHeeledShoe x, HighHeeledShoe y) { ... }** **overloading**
- **Shoe m(Shoe y) { ... }** **overloading**
- **Shoe z(Shoe x, Shoe y) { ... }** **none (new method declaration)**

Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}
class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}
class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}
class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("paddle!"); }
}
```

Given the declarations on the left, determine the outcome of the expressions below.

**Bird b = new Bird();
b.move();**

**Bird b = new Duck();
b.move(42);**

**Duck donald = new RubberDuck();
donald.swim();**

**Bird b = new Canary();
b.move(17);**

**Bird b = new RubberDuck();
b.move(3);**

**Duck donald = new RubberDuck();
donald.move();**

Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}

class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}

class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}

class RubberDuck extends Duck {
    public void speak() { System.out.println("squawk!"); }
    public void move() { speak(); }
    public void swim() { System.out.println("swim!"); }
}
```

Compile error: cannot create instances of an abstract class.

**Bird b = new Bird();
b.move();**

**Bird b = new Canary();
b.move(17);**

**Bird b = new Duck();
b.move(42);**

**Bird b = new RubberDuck();
b.move(3);**

**Duck donald = new RubberDuck();
donald.swim();**

**Duck donald = new RubberDuck();
donald.move();**

Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}

class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}

class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}

class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("paddle!"); }
}
```

Bird
b.move(17);

chirp!
chirp!

Bird b = new Canary();
b.move(17);

Bird b = new Duck();
b.move(42);

Bird b = new RubberDuck();
b.move(3);

Duck donald = new RubberDuck();
donald.swim();

Duck donald = new RubberDuck();
donald.move();

Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}

class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}

class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}

class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("paddle!"); }
}
```

```
Bird b = new Bird();
b.move();
```

```
Bird b = new Canary();
b.move(17);
```

```
Bird b = new Duck();
b.move(42);
```

```
Bird b = new RubberDuck();
b.move(3);
```

flap flap!
quack!

donald.move();

Duck();

Duck();

Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}

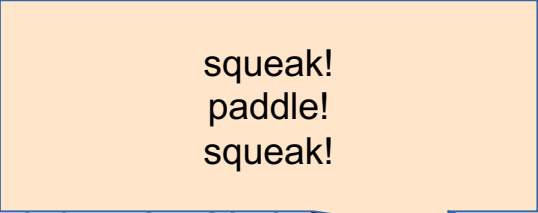
class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}

class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}

class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("paddle!"); }
}
```

Bird b = new Duck();
b.move(42);

Bird b = new RubberDuck();
b.move(17);



squeak!
paddle!
squeak!

Bird b = new Duck();
b.move(42);

Bird b = new RubberDuck();
b.move(3);

Duck donald = new RubberDuck();
donald.swim();

Duck donald = new RubberDuck();
donald.move();

Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}

class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}

class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}

class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("paddle!"); }
}
```

```
Bird b = new Bird();
b.move();
```

```
Bird b = new Canary();
b.move(17);
```

Compile error: no swim method
in class Duck

```
b.move(3);
```

```
Duck();
```

```
Duck donald = new RubberDuck();
donald.swim();
```

```
Duck donald = new RubberDuck();
donald.move();
```

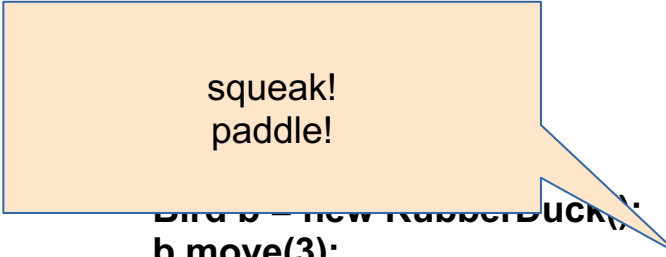
Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}
class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}
class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}
class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("paddle!"); }
}
```

```
Bird b = new Bird();
b.move();
```

```
Bird b = new Canary();
b.move(17);
```

```
Bird b = new RubberDuck();
b.move(3);
```



squeak!
paddle!

```
Duck donald = new RubberDuck();
donald.swim();
```

```
Duck donald = new RubberDuck();
donald.move();
```

Event-Driven Programs

- Sits in an event loop, waiting for events to process
 - often does so until forcibly terminated
- Two common types of event-driven programs:
 - GUIs
 - Web servers
- Where is the event loop in Spark Java?
 - it is created behind the scenes

Design Patterns

- Creational patterns: get around Java constructor inflexibility
 - Sharing: singleton, interning
 - Telescoping constructor fix: builder
 - Returning a subtype: factories
- Structural patterns: translate between interfaces
 - Adapter: same functionality, different interface
 - Decorator: different functionality, same interface
 - Proxy: same functionality, same interface, restrict access
 - All of these are types of **wrappers**

Design Patterns

- Interpreter pattern:
 - Collects code for similar objects, spreads apart code for operations (classes for objects with operations as methods in each class)
 - Easy to add objects, hard to add methods
 - Instance of Composite pattern
- Procedural patterns:
 - Collects code for similar operations, spreads apart code for objects (classes for operations, method for each operand type)
 - Easy to add methods, hard to add objects
 - Ex: Visitor pattern

Design Patterns

Adapter, Builder, Composite, Decorator, Factory, Iterator, Intern, Interpreter, Model-View-Controller (MVC), Observer, Procedural, Prototype, Proxy, Singleton, Visitor, Wrapper

- What pattern would you use to...
 - Remove the addNode/addEdge functionality from your Graph class (throw an UnsupportedOperationException when those methods are called), in order to create an UnmodifiableGraph?
 - We have an existing object that controls a communications channel. We would like to provide the same interface to clients but transmit and receive encrypted data over the existing channel.
 - When the user clicks the “find path” button in the Campus Maps application (HW9), the path appears on the screen.

Design Patterns

Adapter, Builder, Composite, Decorator, Factory, Iterator, Intern, Interpreter, Model-View-Controller (MVC), Observer, Procedural, Prototype, Proxy, Singleton, Visitor, Wrapper

- What pattern would you use to...
 - Remove the addNode/addEdge functionality from your Graph class (throw an UnsupportedOperationException when those methods are called), in order to create an UnmodifiableGraph?
 - **Decorator**
 - We have an existing object that controls a communications channel. We would like to provide the same interface to clients but transmit and receive encrypted data over the existing channel.
 - **Proxy**
 - When the user clicks the “find path” button in the Campus Maps application (HW9), the path appears on the screen.
 - **MVC**
 - **Observer**

Design Patterns

- We have a worksheet with additional practice problems on the website!

Stronger vs Weaker (one more time!)

- In each case, what is the effect of changing the amount of information required about the input?
- Requires more about inputs?
- Promises more about behavior?

Stronger vs Weaker (one more time!)

- In each case, what is the effect of changing the amount of information required about the input?
- Requires more about inputs?

weaker

- Promises more about behavior?

stronger

Stronger vs Weaker

Compared to the spec in the box, what is the effect of using specs A,B,C in terms of our statement's strength (weaker/stronger/neither)?

```
@requires key is a key in this  
@return the value associated with key  
@throws NullPointerException if key is null
```

- A. @requires that key is a key in this and key != null
@return the value associated with key
- B. @return the value associated with key if key is a key in *this*, or null if key is not associated with any value
- C. @return the value associated with key
@throws NullPointerException if key is null
@throws NoSuchElementException if key is not a key *this*

Stronger vs Weaker

Compared to the spec in the box, what is the effect of using specs A,B,C in terms of our statement's strength (weaker/stronger/neither)?

```
@requires key is a key in this  
@return the value associated with key  
@throws NullPointerException if key is null
```

- A. @requires that key is a key in this and key != null
@return the value associated with key **WEAKER**
- B. @return the value associated with key if key is a key in *this*, or null if key is not associated with any value **NEITHER**
- C. @return the value associated with key
@throws NullPointerException if key is null
@throws NoSuchElementException if key is not a key *this* **STRONGER**

Exceptions

- Unchecked exceptions are ignored by the compiler.
- If a method throws a checked exception or calls a method that throws a checked exception, then it must either:
 - catch the exception
 - declare it in `@throws`

Exceptions Examples

Should these be checked or unchecked?

- Attempt to write an invalid type into an array
E.g., write **Double** into **Integer []** cast to **Number []**
- Attempt to open a file that does not exist
- Attempt to create a URL from invalidly formatted text
E.g., “http:/foo” (only one “/”)

Exceptions Examples

Should these be checked or unchecked?

- Attempt to write an invalid type into an array
E.g., write `Double` into `Integer []` cast to `Number []`
unchecked
- Attempt to open a file that does not exist
checked
- Attempt to create a URL from invalidly formatted text
E.g., “http:/foo” (only one “/”)
debatable – could see either one

Thanks for the great quarter!