Advanced Object-Oriented Programming

CPT204 – Lecture 11
Erick Purwanto

CPT204 Advanced Object-Oriented Programming

Lecture 11

Invariant, Abstraction Function, Comparability, Equality

Welcome !

- Welcome to Lecture 11 !

- In this lecture we are going to
  - learn to implement an ADT using *abstraction functions* and *representation invariants*
  - learn about *casting*: what it is and what is it not
  - learn how to make an instance of your data structure can be *compared* to another
  - learn to define the *equality* operation for an ADT

Invariant and Abstraction Function

- In the first part of the lecture, we discuss about Invariant, that we have used before

- We will also going to introduce several ideas:

  - Representation exposure

  - Abstraction functions

  - Representation invariants

- We study a more formal idea of what it means for a class to implement an ADT, via the notions of abstraction functions and rep invariants

  - The abstraction function will give us a way to cleanly define the equality operation on an abstract data type

  - The rep invariant will make it easier to catch bugs caused by a corrupted data structure

Invariant (1)

- Resuming our discussion of what makes a good abstract data type, the final, and perhaps most important, property of a good abstract data type is that **it preserves its own invariants**

- An invariant is a property of a program that is *always true*, for every possible runtime state of the program

- Immutability is one crucial invariant that we've already encountered: once created, an immutable object should always represent the same value, for its entire lifetime

- Saying that the ADT preserves its own invariants means that the ADT is *responsible* for ensuring that its own invariants hold

  - it *doesn't* depend on good behavior from its clients

Invariant (2)

- When an ADT preserves its own invariants, reasoning about the code becomes much easier

  - If you can count on the fact that `Strings` never change, you can rule out that possibility when you're debugging code that uses `Strings` — or when you're trying to establish an invariant for another ADT that uses `Strings`

  - Contrast that with a string type that guarantees that it will be immutable only if its clients promise not to change it

    - then you'd have to check all the places in the code where the string might be used

Immutability (1)

- In this example, we want to have immutability as an invariant:

```java
/**
 * This immutable data type represents a tweet from Twitter.
 */
public class Tweet {

    public String author;
    public String text;
    public Date timestamp;

    /**
     * Make a Tweet.
     * @param author    Twitter user who wrote the tweet
     * @param text      text of the tweet
     * @param timestamp date/time when the tweet was sent
     */
    public Tweet(String author, String text, Date timestamp) {
        this.author = author;
        this.text = text;
        this.timestamp = timestamp;
    }
}
```

Immutability (2)

- How do we guarantee that these Tweet objects are immutable — that, once a tweet is created, its author, message, and date can never be changed?

- The first threat to immutability comes from the fact that clients can — in fact must — directly access its fields

  - So nothing's stopping us from writing code like this:

```
Tweet t = new Tweet("Donald Trump", "Despite the constant negative
                                press covfefe", new Date());
t.author = "abcde";
```

Immutability (3)

- This is a trivial example of **representation exposure**, meaning that code outside the class can modify the representation directly

- Rep exposure like this threatens not only invariants, but also *representation independence*

  - we can't change the implementation of `Tweet` without affecting all the clients who are directly accessing those fields

Immutability (4)

- Fortunately, Java gives us language mechanisms to deal with this kind of rep exposure:

```java
public class Tweet {

    private final String author;
    private final String text;
    private final Date timestamp;

    public Tweet(String author, String text, Date timestamp) {
        this.author = author;
        this.text = text;
        this.timestamp = timestamp;
    }

    /** @return Twitter user who wrote the tweet */
    public String getAuthor() {
        return author;
    }

    /** @return text of the tweet */
    public String getText() {
        return text;
    }

    /** @return date/time when the tweet was sent */
    public Date getTimestamp() {
        return timestamp;
    }

}
```

Immutability (5)

- The `private` and `public` keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class

- The `final` keyword also helps by guaranteeing that the fields of this immutable type *won't be reassigned* after the object is constructed
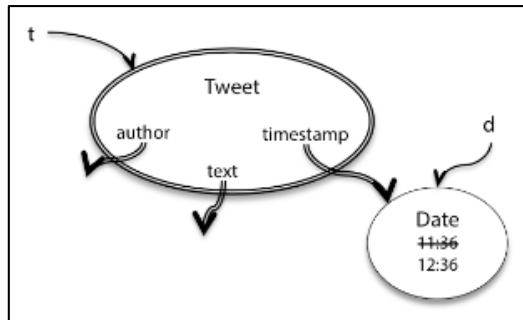
Immutability (6)

- But that's not the end of the story: the rep is still exposed!

- Consider this perfectly reasonable client code that uses Tweet

```
/**
 * @return a tweet that retweets t, one hour later
 */
public static Tweet retweetLater(Tweet t) {
    Date d = t.getTimestamp();
    d.setHours(d.getHours() + 1);
    return new Tweet("abcde", t.getText(), d);
}
```

Immutability (7)

- `retweetLater` takes a tweet and should return another tweet with the same message (called a retweet) but sent an hour later

- The `retweetLater` method might be part of a system that automatically echoes funny things that Twitter celebrities say

- What's the problem here?

- The `getTimestamp` call returns *a reference* to the same `Date` object referenced by tweet `t`

  - `t.timestamp` and `d` are aliases to the same mutable object

  - So when that `Date` object is mutated by `d.setHours()`, this affects the date in `t` as well, as shown in the snapshot diagram

Immutability (8)

- `Tweet`'s immutability invariant has been broken

  - the problem is that `Tweet` *leaked out* a reference to a mutable object that its immutability depended on

- We exposed the rep, in such a way that `Tweet` can no longer guarantee that its objects are immutable

  - perfectly reasonable client code created a subtle bug

- We can patch this kind of rep exposure by using **defensive copying**: making a copy of a mutable object to avoid leaking out references to the rep

Immutability (9)

- Here's the code using defensive copying:

```java
public Date getTimestamp() {
    return new Date(timestamp.getTime());
}
```

- Mutable types often have a **copy constructor** that allows you to make a new instance that duplicates the value of an existing instance

  - In this case, `Date`'s copy constructor uses the timestamp value, measured in milliseconds since January 1, 1970

- As another example, `StringBuilder`'s copy constructor takes a `String`

- Another way to copy a mutable object is `clone()`, which is supported by some types but not all

Immutability (10)

- So we've done some defensive copying in the return value of `getTimestamp`

- But we're not done yet! There's still rep exposure; consider this client code:

```java
/**
 * @return a list of 24 inspiring tweets, one per hour today
 */
public static List<Tweet> tweetEveryHourToday() {
    List<Tweet> list = new ArrayList<Tweet>();
    Date date = new Date();
    for (int i = 0; i < 24; i++) {
        date.setHours(i);
        list.add(new Tweet("abcde", "jiāyóu! you can do it!", date));
    }
    return list;
}
```

Immutability (11)

- The code intends to advance a single `Date` object through the 24 hours of a day, creating a tweet for every hour

- But notice that the constructor of `Tweet` saves the reference that was passed in, so all 24 `Tweet` objects end up with the same time, as shown in this snapshot diagram

  - Again, the immutability of `Tweet` has been violated

- We can fix this problem too by using **defensive copying**, this time in the constructor:
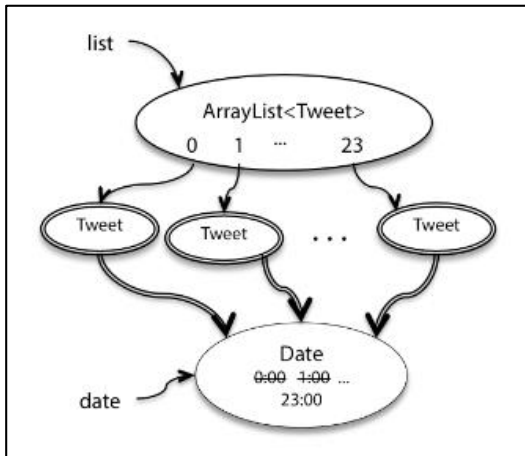

```java
public Tweet(String author, String text, Date timestamp) {
    this.author = author;
    this.text = text;
    this.timestamp = new Date(timestamp.getTime());
}
```

Immutability (12)

- In general, you should carefully inspect the argument types and return types of all your ADT operations

    - If any of the types are mutable, make sure your implementation doesn't return direct references to its representation

    - Doing that creates rep exposure!

- You may object that this seems wasteful

    - Why make all these copies of dates?

- Why can't we just solve this problem by a carefully written specification?

Immutability (13)

- Why **can't** we just solve this problem by a carefully written specification, like this:

```
/**
 * Make a Tweet.
 * @param author    Twitter user who wrote the tweet
 * @param text      text of the tweet
 * @param timestamp date/time when the tweet was sent.
 *         Caller must never mutate this Date object again!
 */
public Tweet(String author, String text, Date timestamp) {
```

Immutability (14)

- This approach is sometimes taken when there isn't any other reasonable alternative — for example, when the mutable object is too large to copy efficiently

  - But the cost in your ability to reason about the program, and your ability to avoid bugs, is enormous

  - In the absence of compelling arguments to the contrary, it's almost always worth it for an abstract data type to guarantee its own invariants, and preventing rep exposure is essential to that

- An even better solution is to prefer immutable types

  - If — as recommended in Mutability and Immutability Groundhog Day example in Lecture 9 — we had used an immutable date object, like `java.time.ZonedDateTime`, instead of the mutable `java.util.Date`, then we would have ended this part of lecture after talking about `public` and `private`

  - **No** further rep exposure would have been possible

Immutable Wrappers Around Mutable Data Types

- The Java collections classes offer an interesting compromise: **immutable wrappers**

  - we discussed about these before in the lecture on Mutability and Immutability

- `Collections.unmodifiableList()` takes a (mutable) `List` and wraps it with an object that looks like a `List`, but whose mutators are disabled — `set()`, `add()`, `remove()` throw exceptions

  - so you can construct a list using mutators, then seal it up in an unmodifiable wrapper (*and throw away* your reference to the original mutable list), and get an immutable list

- The downside here is that *you get immutability at runtime*, but **not** at compile time

  - Java won't warn you at compile time if you try to `sort()` this unmodifiable list

  - you'll just get an exception at runtime

  - but that's still better than nothing, so using unmodifiable lists, maps, and sets can be a very good way to reduce the risk of bugs

Abstract Space vs Rep Space (1)

- We now take a deeper look at the theory underlying abstract data types

  - This theory is not only elegant and interesting in its own right; it also has immediate practical application to the design and implementation of abstract types

  - If you understand the theory deeply, you'll be able to build better abstract types, and will be less likely to fall into subtle traps

- In thinking about an abstract type, it helps to consider the relationship between two spaces of values

- **The space of <u>abstract values</u>** consists of the values that the type is *designed to support*, from the client's point of view

  - for example, an abstract type for unbounded integers, like Java's `BigInteger`, would have the mathematical integers as its abstract value space
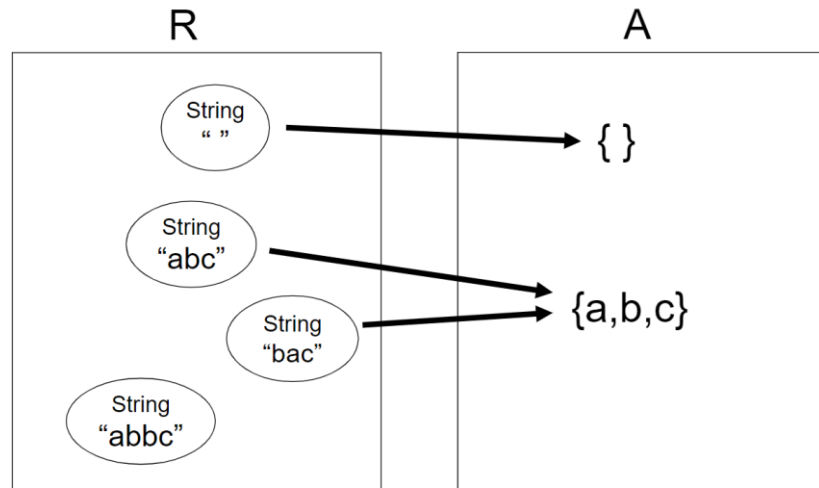
Abstract Space vs Rep Space (2)

- **The space of representation values** (or **rep values** for short) consists of *the Java objects that actually implement* the abstract values
  - for example, a `BigInteger` value might be implemented using *an array of digits*, represented as primitive int values
  - The rep space would then be *the set of all such arrays*
- In simple cases, an abstract type will be implemented as *a single Java object*, but more commonly a small network of objects is needed
  - for example, a rep value for `List` might be a linked list, a group of objects linked together by next and previous pointers; so a rep value is not necessarily a single object, but often something rather complicated
- Now of course the implementer of the abstract type must be interested in the representation values, since it is the implementer's job to *achieve the illusion* of the abstract value space using the rep value space

Abstract Space vs Rep Space (3)

- Suppose, for example, that we choose to use a *string* to represent *a set of characters*:

```
public class CharSet {
    private String s;
    ...
}
```

- Then the **rep space R** contains *Strings*, and the **abstract space A** is *mathematical sets of characters*

- We can show the two value spaces graphically, with an arc from a rep value to the abstract value it represents
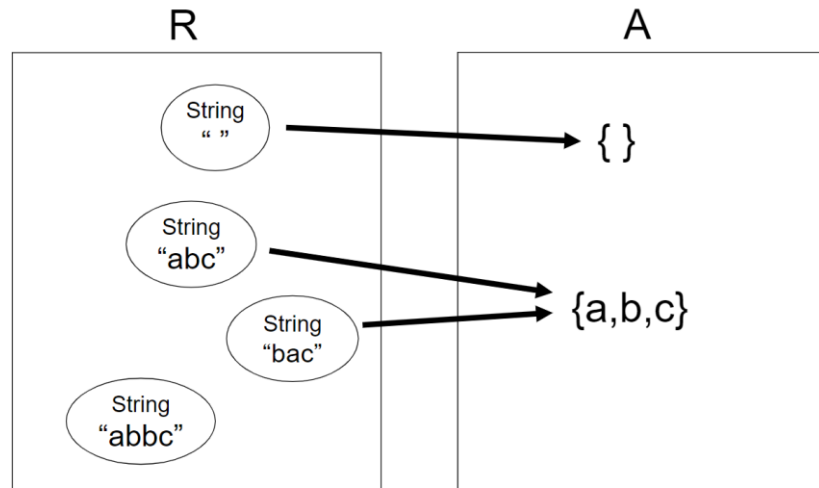
Abstract Space vs Rep Space (4)

There are several things to note about this picture:

- **Every abstract value is mapped to by some rep value**
  The purpose of implementing the abstract type is to support operations on abstract values;
 Presumably, then, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable

Abstract Space vs Rep Space (5)

- **Some abstract values are mapped to by more than one rep value**
  This happens because the representation isn't a tight encoding; There's more than one way to represent an unordered set of characters as a string
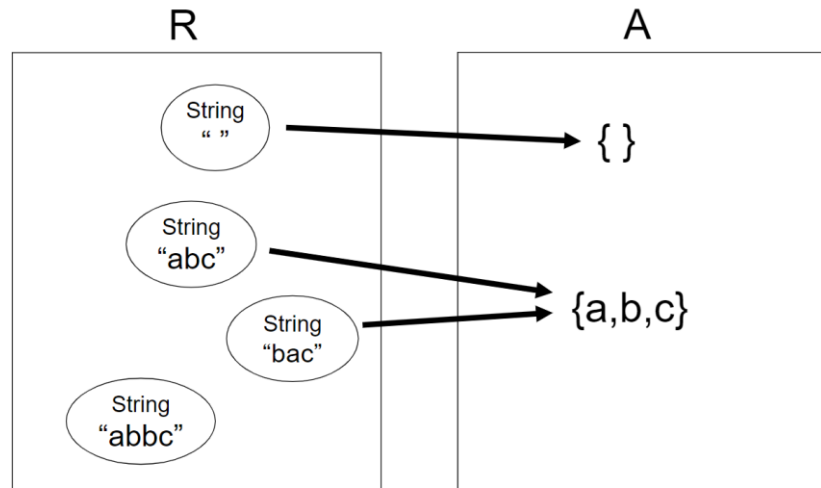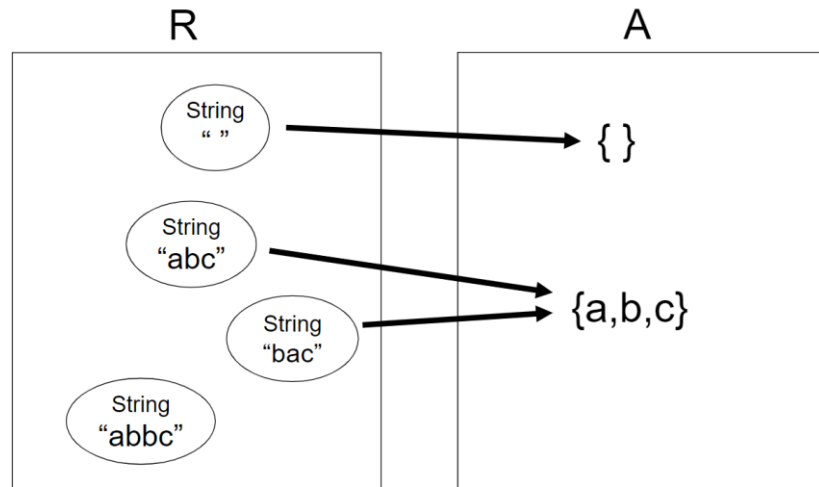
Abstract Space vs Rep Space (6)

- **Not all rep values are mapped**
  Notice the string "abbc" is not mapped;
  In this case, we have decided that the string should not contain duplicates;
  This will allow us to terminate the ʀᴇᴍᴏᴠᴇ method when we hit the first instance of a particular character, since we know there can be at most one

Rep Invariant and Abstraction Function (1)

In practice, we can only illustrate a few elements of the two spaces and their relationships; the graph as a whole is infinite

So we describe it by giving two things:

- An **abstraction function** that maps rep values to the abstract values they represent:

    AF : R → A

    The *arcs* in the diagram show the abstraction function
    In the terminology of functions, the properties we discussed above can be expressed by saying that the function is *surjective* (also called onto), *not necessarily injective* (also called one-to-one), therefore *not necessarily bijective*;
    and often *partial*

> how to interpret rep values as abstract values

Rep Invariant and Abstraction Function (2)

- A **rep invariant** that maps rep values to booleans:

    RI : R → boolean

    For a rep value r, $RI(r)$ is true if and only if r is mapped by AF

    which rep values are legal

    In other words, RI tells us whether a given rep value is well-formed

    Alternatively, you can think of RI as *a set*: it's the subset of rep values on which AF is defined

Rep Invariant and Abstraction Function Example 1 (1)

- For example, the diagram at the right showing a rep for CharSet that forbids repeated characters
  - *RI("a")* = true, *RI("ac")* = true and *RI("acb")* = true
 - but *RI("aa")* = false and *RI("abbc")* = false
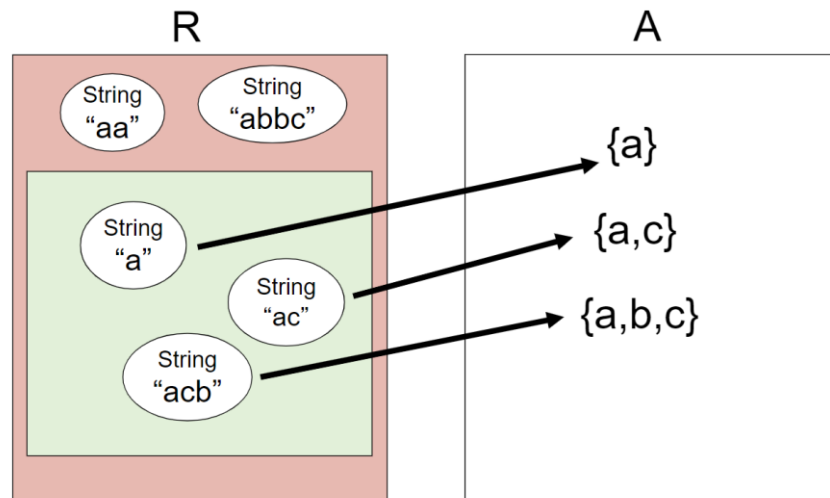
- Rep values that *obey* the rep invariant are shown in the green part of the R space, and must map to an abstract value in the A space

- Rep values that *violate* the rep invariant are shown in the red zone, and have no equivalent abstract value in the A space

Rep Invariant and Abstraction Function Example 1 (2)

- Both the rep invariant and the abstraction function should be documented in the code, right next to the declaration of the rep itself:

```java
public class CharSet {

    private String s;

    // Rep invariant:

    //  s contains no repeated characters

    // Abstraction function:

    //  AF(s) = {s[i] | 0 <= i < s.length()}

    ...

}
```
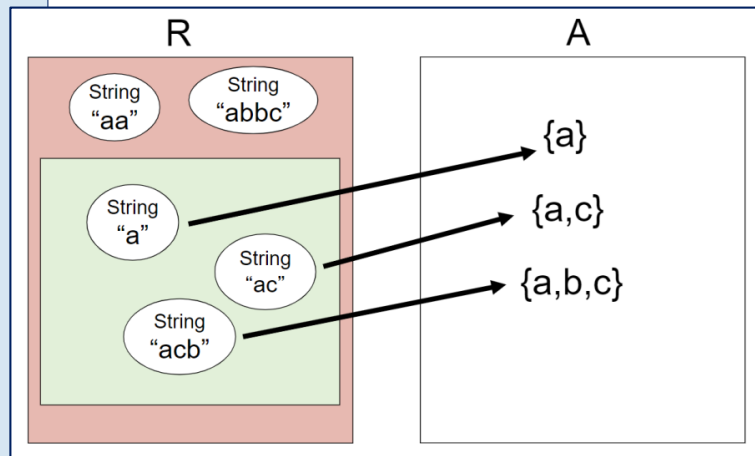
Rep Value Spaces, Rep Invariant, Abstraction Function

- A *common confusion* about abstraction functions and rep invariants is that they are determined by the choice of rep and abstract value spaces, or even by the abstract value space alone
  - if this were the case, they would be of little use, since they would be saying something redundant that's already available elsewhere
- The abstract value space *alone* **doesn't** *determine* AF or RI: there can be ***several*** representations for the ***same*** abstract type
  - A set of characters could equally be represented as *a string*, as above, or as *a bit vector*, with one bit for each possible character
  - Clearly we need two *different* abstraction functions to map these two different rep value spaces
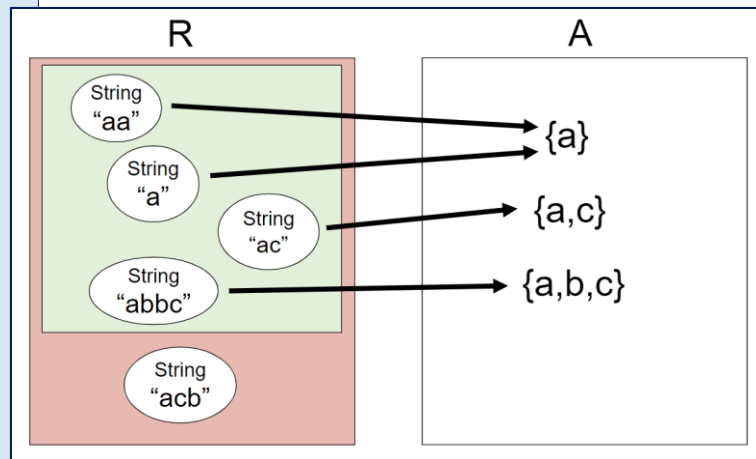
Rep Invariant and Abstraction Function Example 2 (1)

- It's less obvious why the choice of both spaces doesn't determine AF and RI

  - The key point is that defining a type for the rep, and thus choosing the values for the space of rep values, does *not* determine *which of the rep values will be deemed to be legal*, and of those that are legal, *how they will be interpreted*

  - Rather than deciding, as we did above, that the strings have no duplicates, we could instead *allow duplicates*, but at the same time require that the characters be *sorted*, appearing in nondecreasing order

    - This would allow us to perform a binary search on the string and thus check membership in logarithmic rather than linear time

● Same rep value space — different rep invariant:

```
public class CharSet {

    private String s;

    // Rep invariant:

    //   s[0] <= s[1] <= ... <= s[s.length()-1]

    // Abstraction function:

    //   AF(s) = {s[i] | 0 <= i < s.length()}

        ...

}
```
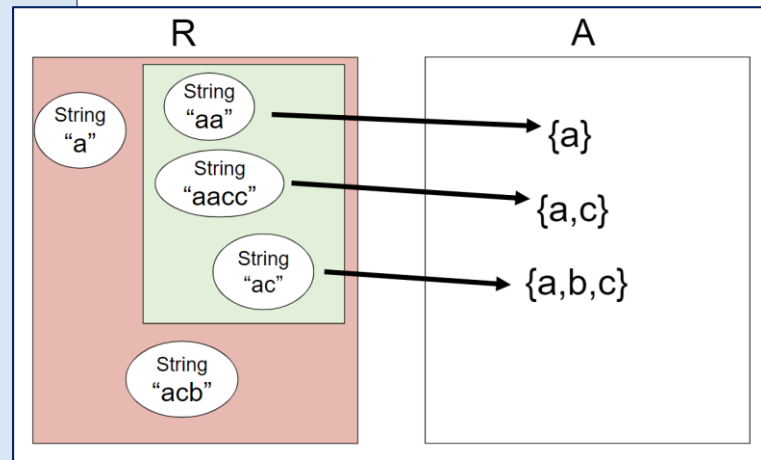
Rep Invariant and Abstraction Function Example 3 (1)

- Even with the *same type* for the rep value space and the same rep invariant RI, we might still interpret the rep differently, with *different* abstraction functions AF

- Suppose RI admits any string of characters

- Then we could define AF, as above, to interpret the array's elements as the elements of the set

- But there's no a priori reason to let the rep decide the interpretation

  - Perhaps we'll interpret consecutive pairs of characters as subranges, so that the string rep "acgg" is interpreted as two range pairs, [a-c] and [g-g], and therefore represents the set {a, b, c, g}

Rep Invariant and Abstraction Function Example 3 (2)

- Here's what the AF and RI would look like for that representation:

```
public class CharSet {

    private String s;

    // Rep invariant:

    //    s.length() is even

    //    s[0] <= s[1] <= ... <= s[s.length()-1]

    // Abstraction function:

    //    represents the union of the ranges

    //    {s[i]...s[i+1]} for each adjacent pair

    //    of characters in s

    ...

}
```

In-Class Quiz 1.1

- Consider the rep of `CharSet` in Example 3:

```java
public class CharSet {
    private String s;
    // Rep invariant:
    //    s.length() is even
    //    s[0] <= s[1] <= ... <= s[s.length()-1]
    // Abstraction function:
    //    represents the union of the ranges
    //    {s[i]...s[i+1]} for each adjacent pair
    //    of characters in s
    ...
}
```


- Which of the following values of s satisfy this rep invariant?

  ☐   "abc"   ☐   "ad"

  ☐   "abcd"   ☐   "adad"

  ☐   "eeee"   ☐   ""

In-Class Quiz 1.2

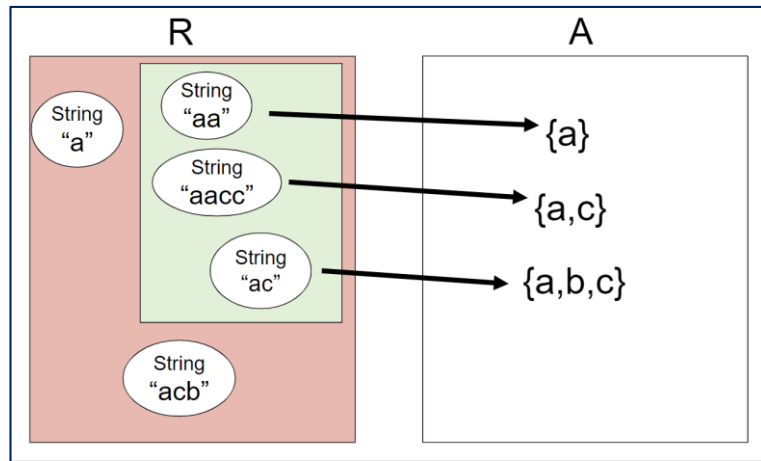- Consider the rep of `CharSet` in Example 3:

```java
public class CharSet {
    private String s;
    // Rep invariant:
    //    s.length() is even
    //    s[0] <= s[1] <= ... <= s[s.length()-1]
    // Abstraction function:
    //    represents the union of the ranges
    //    {s[i]...s[i+1]} for each adjacent pair
    //    of characters in s
    ...
}
```


- Which of the following does AF("acfg") map to?

    ○ {a,b,c,d,e,f,g}

    ○ {a,b,c,f,g}

    ○ {a,c,f,g}

    ○ some other abstract value

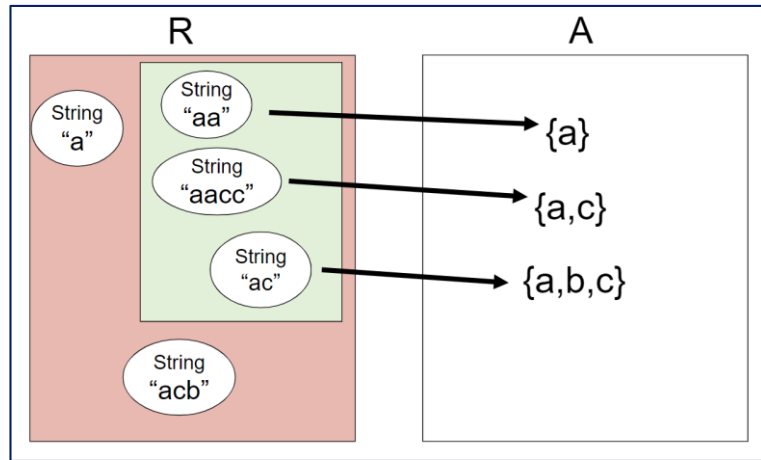   ○ no abstract value, because "acfg" does not satisfy the rep invariant

In-Class Quiz 1.3

- Consider the rep of `CharSet` in Example 3:

```java
public class CharSet {
    private String s;
    // Rep invariant:
    //   s.length() is even
    //   s[0] <= s[1] <= ... <= s[s.length()-1]
    // Abstraction function:
    //   represents the union of the ranges
    //   {s[i]...s[i+1]} for each adjacent pair
    //   of characters in s
    ...
}
```
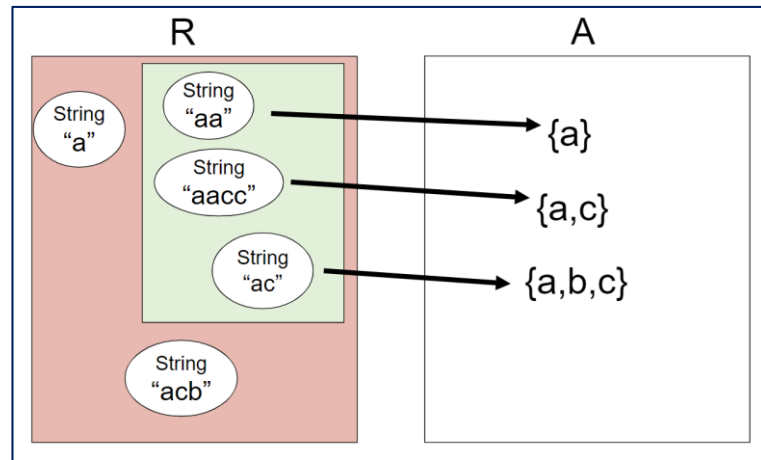

- Which of these values does the abstraction function map to the same abstract value as it maps "tv"?

- ❑ "ttv"
- ❑ "ttuv"
- ❑ "ttuuvv"
- ❑ "tuv"

Choosing Rep Invariant and Abstraction Function

- The essential point is that designing an abstract type means **not only *choosing the two spaces*** — the abstract value space for the specification and the rep value space for the implementation — **but also deciding *which rep values are legal*** and ***how to interpret them* as abstract values**

- It's critically important to write down these assumptions in your code, as we've done above, so that future programmers (and your future self) are aware of what the representation actually means

Example: Rational Numbers (1)

- Here's an example of an ADT for rational numbers with rep invariant and abstraction function

```java
public class RatNum {
    private final int numerator;
    private final int denominator;
    // Rep invariant:
    //   denom > 0
    //   numer/denom is in reduced form
    // Abstraction Function:
    //   represents rational nbr numer/denom

    /**
     * Make a new Ratnum == n.
     * @param n value
     */
    public RatNum(int n) {
        numerator = n;
        denominator = 1;
        checkRep();
    }
}
```
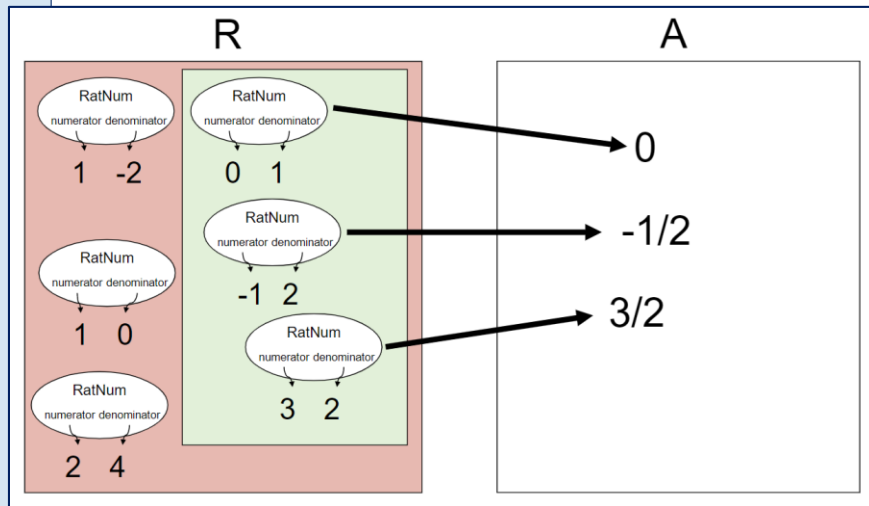
Example: Rational Numbers (2)

```java
/**
 * Make a new RatNum == (n / d).
 * @param n numerator
 * @param d denominator
 * @throws ArithmeticException if d == 0
 */
public RatNum(int n, int d) throws ArithmeticException {
    // reduce ratio to lowest terms
    int g = gcd(n, d);
    n = n / g;
    d = d / g;
    // make denominator positive
    if (d < 0) {
        numerator = -n;
        denominator = -d;
    } else {
        numerator = n;
        denominator = d;
    }
    checkRep();
}
}
```

Example: Rational Numbers (3)

- Here we illustrate the abstraction function and rep invariant for the code

- The RI requires that numerator/denominator pairs be in reduced form (i.e., lowest terms), so pairs like (2,4) and (18,12) should be drawn as outside the RI

representation type R

abstract type A

(-1,-2)   (0,1)   0
                           1/2
(1,0)   (1,2)
(2,4)   (3,2)   3/2

rep invariant RI
describes legal reps

abstraction function AF
maps legal reps to abstract values

abstraction barrier preserves
representation independence
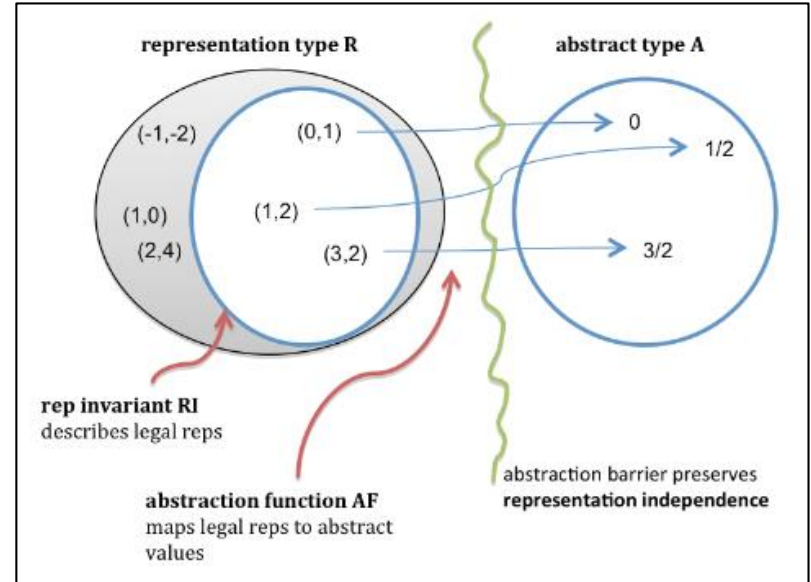
Checking the Rep Invariant (1)

- The rep invariant isn't just a neat mathematical idea

  - If your implementation asserts the rep invariant at run time, then you can catch bugs early

- Here's a method for RatNum that tests its rep invariant:

```java
// Check that the rep invariant is true
// *** Warning: this does nothing unless you turn on assertion checking
//      by running Java with -enableassertions or -ea
private void checkRep() {
    assert denominator > 0;
    assert gcd(Math.abs(numerator), denominator) == 1;
}
```

Checking the Rep Invariant (2)

- You should certainly call `checkRep()` to assert the rep invariant at the end of **every** operation that creates or mutates the rep — in other words, creators, producers, and mutators

  - Look back at the `RatNum` code above, and you'll see that it calls `checkRep()` at the end of **both** constructors

- Observer methods don't normally need to call `checkRep()`, but it's good defensive practice to do so anyway

  - Why? Calling `checkRep()` in every method, including observers, means you'll be more likely to catch rep invariant violations caused by rep exposure

Why is checkRep private?
Who should be responsible for checking and enforcing a rep invariant — clients, or the implementation itself?

No Null Values in the Rep (1)

- Recall from the Lecture 8 that null values are troublesome and unsafe, so much so that we try to remove them from our programming entirely

  - in this course, the preconditions and postconditions of our methods implicitly require that objects and arrays be non-null

- We **extend** that prohibition to the reps of abstract data types

  - by default, the rep invariant implicitly includes `x != null` for every reference `x` in the rep that has object type (including references inside arrays or lists)

  - so if your rep is:

    ```
    class CharSet {
        String s;
    }
    ```

  - then its rep invariant automatically includes `s != null`, and you don't need to state it in a rep invariant comment

No Null Values in the Rep (2)

- When it's time to implement that rep invariant in a `checkRep()` method, however, you still must implement the `s != null` check, and make sure that your `checkRep()` correctly fails when `s` is `null`

- Often that check comes for free from Java, because checking other parts of your rep invariant will throw an exception if `s` is `null`

  - For example, if your `checkRep()` looks like this:

    ```java
    private void checkRep() {
        assert s.length() % 2 == 0;

        ...
    }
    ```

  - then you **don't** need `assert s != null`, because the call to `s.length()` **will fail** just as effectively on a null reference

  - But if `s` is not otherwise checked by your rep invariant, then `assert s != null` explicitly

Documenting the AF, RI, and Safety from Rep Exposure (1)

- It's good practice to document the abstraction function and rep invariant in the class, using comments right where the private fields of the rep are declared
  - we've been doing that above
- Another piece of documentation that you need to write is **a rep exposure safety argument**
  - this is a comment that examines each part of the rep, looks at the code that handles that part of the rep (particularly with respect to parameters and return values from clients, because that is where rep exposure occurs), and presents a reason why the code doesn't expose the rep

Documenting the AF, RI, and Safety from Rep Exposure (2)

- Here's an example of Tweet with its rep invariant, abstraction function, and safety from rep exposure fully documented:

```java
// Immutable type representing a tweet.
public class Tweet {

    private final String author;
    private final String text;
    private final Date timestamp;

    // Rep invariant:
    //    author is a Twitter username (a nonempty string of letters, digits, underscores)
    //    text.length <= 280
    // Abstraction Function:
    //    represents a tweet posted by author, with content text, at time timestamp
    // Safety from rep exposure:
    //    All fields are private;
    //    author and text are Strings, so are guaranteed immutable;
    //    timestamp is a mutable Date, so Tweet() constructor and getTimestamp()
    //        make defensive copies to avoid sharing the rep's Date object with clients.
```

Documenting the AF, RI, and Safety from Rep Exposure (3)

```
    public Tweet(String author, String text, Date timestamp) { ... }

    public String getAuthor() { ... }

    public String getText() { ... }

    public Date getTimestamp() { ... }

}
```

- Notice that we don't have any explicit rep invariant conditions on timestamp (aside from the conventional assumption that `timestamp != null`, which we have for all object references)

- But we still need to include timestamp in the rep exposure safety argument, because the immutability property of the whole type depends on all the fields remaining unchanged

Documenting the AF, RI, and Safety from Rep Exposure (4)

- Compare the argument above with an example of a broken argument involving mutable `Date` objects:

```java
public class Timespan {

    private final Date start;
    private final Date end;

    // Rep invariant:
    //    !end.before(start)
    // Abstraction Function:
    //    represents the time interval from start to end, inclusive
    // Safety from rep exposure:
    //    All fields are private and immutable. (<== oops, false! Date is mutable)

        ...
}
```

Documenting the AF, RI, and Safety from Rep Exposure (5)

- Here are the arguments for RatNum

```java
// Immutable type representing a rational number.
public class RatNum {
    private final int numer;
    private final int denom;

    // Rep invariant:
    //    denom > 0
    //    numer/denom is in reduced form, i.e. gcd(|numer|,denom) = 1
    // Abstraction Function:
    //    represents the rational number numer / denom
    // Safety from rep exposure:
    //    All fields are private, and all types in the rep are immutable.

    public RatNum(int n) { ... }
    public RatNum(int n, int d) throws ArithmeticException { ... }
    ...
}
```

- Notice that an immutable rep is particularly easy to argue for safety from rep exposure

How to Establish Invariants (1)

- An invariant is a property that is true for the entire program — which in the case of an invariant about an object, reduces to the entire lifetime of the object

- To make an invariant hold, we need to:

    - **make the invariant true in the initial state of the object**;   and

    - **ensure that all changes to the object keep the invariant true**

- Translating this in terms of the types of ADT operations, this means:

    - creators and producers must ***establish the invariant*** for new object instances;   and

    - mutators and observers must ***preserve the invariant***

How to Establish Invariants (2)

- The risk of rep exposure makes the situation more complicated;
  If the rep is exposed, then the object might be changed anywhere in the program,
  not just in the ADT's operations, and we can't guarantee that the invariant still
  holds after those arbitrary changes

So the full rule for proving invariants is: (called **structural induction**)

- if an invariant of an abstract data type is

  - established by creators and producers;

  - preserved by mutators, and observers;   and

  - no representation exposure occurs,

  then the invariant is true of all instances of the abstract data type

ADT invariants replace preconditions (1)

- Now let's bring a lot of pieces together

- An enormous advantage of a well-designed abstract data type is that it encapsulates and enforces properties that we would otherwise have to stipulate in a precondition

- For example, instead of a spec like this, with an elaborate precondition:

```
/**
 * @param set1 is a sorted set of characters with no repeats
 * @param set2 is likewise
 * @return characters that appear in one set but not the other,
 *         in sorted order with no repeats
 */
static String exclusiveOr(String set1, String set2);
```

ADT invariants replace preconditions (2)

- We can instead use an ADT that captures the desired property:

```java
/** @return characters that appear in one set but not the other */
static SortedSet<Character> exclusiveOr(SortedSet<Character> set1,
                                        SortedSet<Character> set2);
```

- This is easier to understand, because the *name of the ADT* conveys all the programmer needs to know

- It's also safer from bugs, because Java static checking comes into play, and the required condition (sorted with no repeats) can be enforced in exactly one place, the SortedSet type

- Many of the places where we used preconditions on the problem sets would have benefited from a custom ADT instead

Inheritance, DMS, Comparability, Equality

- For the second part of the lecture, we will continue our discussion last week on

  - Inheritance

  - Dynamic Method Selection

    - also known as Polymorphism

  - Comparability

  - Equality

Recall from last week: Which Method Selected

- Recall that if X is a superclass of Y, then an X variable can hold a reference to a Y

```java
public static void main(String[] args) {
    ListIF<String> list2 = new SLList<String>();
    list2.addLast("abc");
    list2.print();
}
```

- Which print method will run when the code above executes?

  - SLList.print(), and not ListIF.print()

  - How does it work?

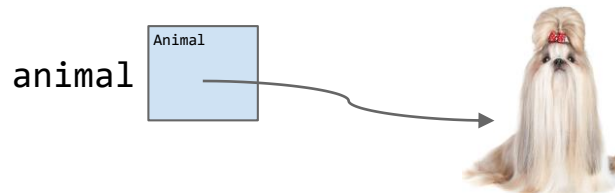   - Before we can answer, we need to understand static and dynamic type

Recall from last week: Static Type vs Dynamic Type

- Every variable in Java has a ***compile-time*** type = **static** type

    - This is the type specified at declaration and it *never* changes!

- Variables also have a ***run-time*** type = **dynamic** type

    - This is the type specified *at instantiation* (e.g. when using new)

    - Equal to the type of the object *being* pointed at

very important
to remember !

```java
public static void main(String[] args) {
    Animal animal;
    animal = new Dog();
}
```

animal

Animal

variable animal has:
static type = Animal
dynamic type = Dog

Recall from last week: Static Type vs Dynamic Type

- Every variable in Java has a *compile-time* type = **static** type

  - This is the type specified at declaration and it *never* changes!

- Variables also have a *run-time* type = **dynamic** type

  - This is the type specified *at instantiation* (e.g. when using new)

  - Equal to the type of the object *being* pointed at

very important
to remember !

```java
public static void main(String[] args) {
    Animal animal;
    animal = new Dog();
    animal = new Pig();
}
```

animal

Animal

variable animal has:
static type = Animal
dynamic type = Pig
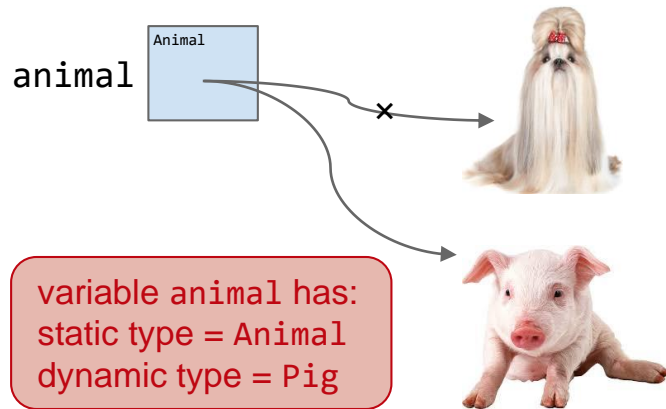
Recall from last week: DMS for Overridden Methods

- Suppose we call a method `m()` of an object using a variable with:

    ○ static type `X` and dynamic type `Y`

- First, the compiler records the X's method `m()`, to be used in run-time

- At run-time, **if** Y overrides the method `m()`, **then** Y's method `m()` is used *instead*

    ○ this is known as **dynamic method selection**

Recall from last week: DMS for Print()

- Suppose we call a method `m()` of an object using a variable with:

  - static type `X` and dynamic type `Y`

- First, the compiler records the `X`'s method `m()`, to be used in run-time

- At run-time, **if** `Y` overrides the method `m()`, **then** `Y`'s method `m()` is used *instead*

  - this is known as **dynamic method selection**

- Therefore, `print()` that belongs to `SLList` is used:

```java
public static void main(String[] args) {
    ListIF<String> list2 = new SLList<String>();
    list2.addLast("abc");
    list2.print();
}
```

Dynamic Method Selection for Overridden Methods (1)

- Suppose we call a method `m()` of an object using a variable with:

  - static type `X` and dynamic type `Y` ← Y is a subclass of X

  - that is: `X varName = new Y();`
                    `varName.m();`

  if X does **not** define method m(),
  (even if Y defines m())
  it will cause a compile error

- First, the compiler records the X's method `m()`, to be used in run-time

- At run-time, **if** Y overrides the method `m()`, **then** Y's method `m()` is used *instead*

  - this is known as **dynamic method selection**

  if Y does **not** override m(),
  (even if Y overload m)
  X.m() will be used

Dynamic Method Selection for Overridden Methods (2)

- Suppose we call a method `m()` of an object using a variable with:

  - static type `X` and dynamic type `Y`

  - that is: `X varName = new Y();`

    `varName.m();`

- First, the compiler records the X's method `m()`, to be used in run-time

- At run-time, **if** Y overrides the method `m()`, **then** Y's method `m()` is used *instead*

  - this is known as **dynamic method selection**

Thus, DMS happens if:
1. X defines m()
2. Y overrides m()

Another Example

- Recall from last week, that `RecoSLList` is a subclass of `SLList`
  - which line causes error, and if not, whose method is called?

```java
public static void main(String[] args) {

    RecoSLList<Integer> rsl =
        new RecoSLList<Integer>(5);
    SLList<Integer> sl = rsl;

    sl.addLast(10);
    sl.delLast();

    sl.printDelItems();

    RecoSLList<Integer> rsl2 = sl;
}
```

variable rsl has static type RecoSLList and dynamic type RecoSLList

variable sl has static type SLList and dynamic type RecoSLList, allowed because RecoSLList is a subclass of SLList

Another Example

- Recall from last week, that `RecoSLList` is a subclass of `SLList`
  - which line causes error, and if not, whose method is called?

```java
public static void main(String[] args) {

    RecoSLList<Integer> rsl =
        new RecoSLList<Integer>(5);
    SLList<Integer> sl = rsl;

    sl.addLast(10);
    sl.delLast();

    sl.printDelItems();

    RecoSLList<Integer> rsl2 = sl;
}
```

rsl   static type: RecoSLList
        dynamic type: RecoSLList

sl   static type: SLList
        dynamic type: RecoSLList

RecoSLList does not override
addLast, use SLList.addLast

Another Example

- Recall from last week, that `RecoSLList` is a subclass of `SLList`
  - which line causes error, and if not, whose method is called?

```java
public static void main(String[] args) {

    RecoSLList<Integer> rsl =
        new RecoSLList<Integer>(5);
    SLList<Integer> sl = rsl;

    sl.addLast(10);
    sl.delLast();

    sl.printDelItems();

    RecoSLList<Integer> rsl2 = sl;
}
```

rsl   static type: RecoSLList
         dynamic type: RecoSLList

sl   static type: SLList
        dynamic type: RecoSLList

RecoSLList overrides delLast,
use RecoSLList.delLast

Another Example

- Recall from last week, that `RecoSLList` is a subclass of `SLList`
  - which line causes error, and if not, whose method is called?

```
public static void main(String[] args) {

    RecoSLList<Integer> rsl =
        new RecoSLList<Integer>(5);
    SLList<Integer> sl = rsl;

    sl.addLast(10);
    sl.delLast();

    sl.printDelItems();

    RecoSLList<Integer> rsl2 = sl;
}
```

rsl   static type: RecoSLList
        dynamic type: RecoSLList

sl   static type: SLList
        dynamic type: RecoSLList

compiler first checks that sl
static type SLList does **not**
have printDelItems method,
this will cause a **compile error**!

even though its dynamic type
RecoSLList has printDelItems

Another Example

- Recall from last week, that `RecoSLLList` is a subclass of `SLList`
  - which line causes error, and if not, whose method is called?

```java
public static void main(String[] args) {

    RecoSLLList<Integer> rsl =
        new RecoSLLList<Integer>(5);
    SLList<Integer> sl = rsl;

    sl.addLast(10);
    sl.delLast();

    sl.printDelItems();

    RecoSLLList<Integer> rsl2 = sl;
}
```

rsl   static type: RecoSLLList
         dynamic type: RecoSLLList

sl   static type: SLList
        dynamic type: RecoSLLList

once again, compiler checks based on static type of sl, and because its static type is SLList which is **not** a subclass of RecoSLLList, this will also cause a **compile error**!

compiler plays safe by allowing assignment based on static type

Expressions have Static Types

- Method calls have compile-time/static type equal to their declared type

  - for example, we have `ShihTzu` a subclass of `Dog`, `Dog` has a name and a weight, and the method `public static Dog largerDog(Dog d1, Dog d2)` returns a `Dog` that has a larger weight

  - we call the method on two `ShihTzu`s:

```java
ShihTzu baobei = new ShihTzu("Baobei", 5);
ShihTzu jiaozi = new ShihTzu("Jiaozi", 7);

Dog largerDog = largerDog(baobei, jiaozi);
ShihTzu largerShihTzu = largerDog(baobei, jiaozi);
```

Expressions have Static Types

- Method calls have compile-time/static type equal to their declared type
  - for example, we have `ShihTzu` a subclass of `Dog`, `Dog` has a name and a weight, and the method `public static Dog largerDog(Dog d1, Dog d2)` returns a `Dog` that has a larger weight
  - we call the method on two `ShihTzu`s:

```
ShihTzu baobei = new ShihTzu("Baobei", 5);
ShihTzu jiaozi = new ShihTzu("Jiaozi", 7);

Dog largerDog = largerDog(baobei, jiaozi);
ShihTzu largerShihTzu = largerDog(baobei, jiaozi);
```

the static type of largerDog is Dog, just as declared

Expressions have Static Types

- Method calls have compile-time/static type equal to their declared type

  - for example, we have `ShihTzu` a subclass of `Dog`, `Dog` has a name and a weight, and the method `public static Dog largerDog(Dog d1, Dog d2)` returns a `Dog` that has a larger weight

  - we call the method on two `ShihTzu`s:

```
ShihTzu baobei = new ShihTzu("Baobei", 5);
ShihTzu jiaozi = new ShihTzu("Jiaozi", 7);

Dog largerDog = largerDog(baobei, jiaozi);
ShihTzu largerShihTzu = largerDog(baobei, jiaozi);
```
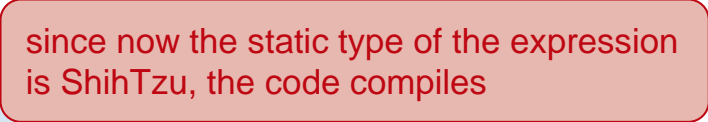
even though the run-time type will be ShihTzu, this will cause a **compile error**, because the static type of largerDog is Dog, which is not ShihTzu, nor a subclass of ShihTzu

Casting

- To specify the static type of any expression, we use **casting** ← we've used it in ARDeque

  - `largerDog(baobei, jiaozi)` has static type `Dog`

  - `(ShihTzu) largerDog(baobei, jiaozi)` has static type `ShihTzu`

    tell compiler the type of the expression

since now the static type of the expression is ShihTzu, the code compiles

```
ShihTzu baobei = new ShihTzu("Baobei", 5);
ShihTzu jiaozi = new ShihTzu("Jiaozi", 7);

ShihTzu largerShihTzu = (ShihTzu)largerDog(baobei, jiaozi);
```

Casting is Dangerous
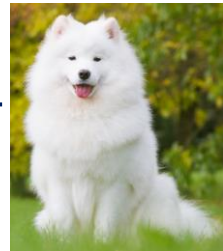
- Casting tells the compiler to *ignore* its type checking duties

  - for example, suppose you also have `Samoyed`, another subclass of `Dog`

  - and say its object has a larger weight

- The result of the method at runtime still has type `Samoyed`!

  - Casting does **not** change it!

```
ShihTzu baobei = new ShihTzu("Baobei", 5);
Samoyed xiaxue = new Samoyed("Xiaxue", 20);


ShihTzu largerShihTzu = (ShihTzu)largerDog(baobei, xiaxue);
```

the compiler is told to pretend to see the expression has type ShihTzu, so it will **allow** this **at compile-time**

when you run the code, **at runtime** you will get **ClassCastException**, and effectively break the type checking we discussed in earlier weeks

use Casting with great care!

The Object Class (1)

- Every type in Java is a subclass of the `Object` class
  - `SLList` implicitly extends `Object`
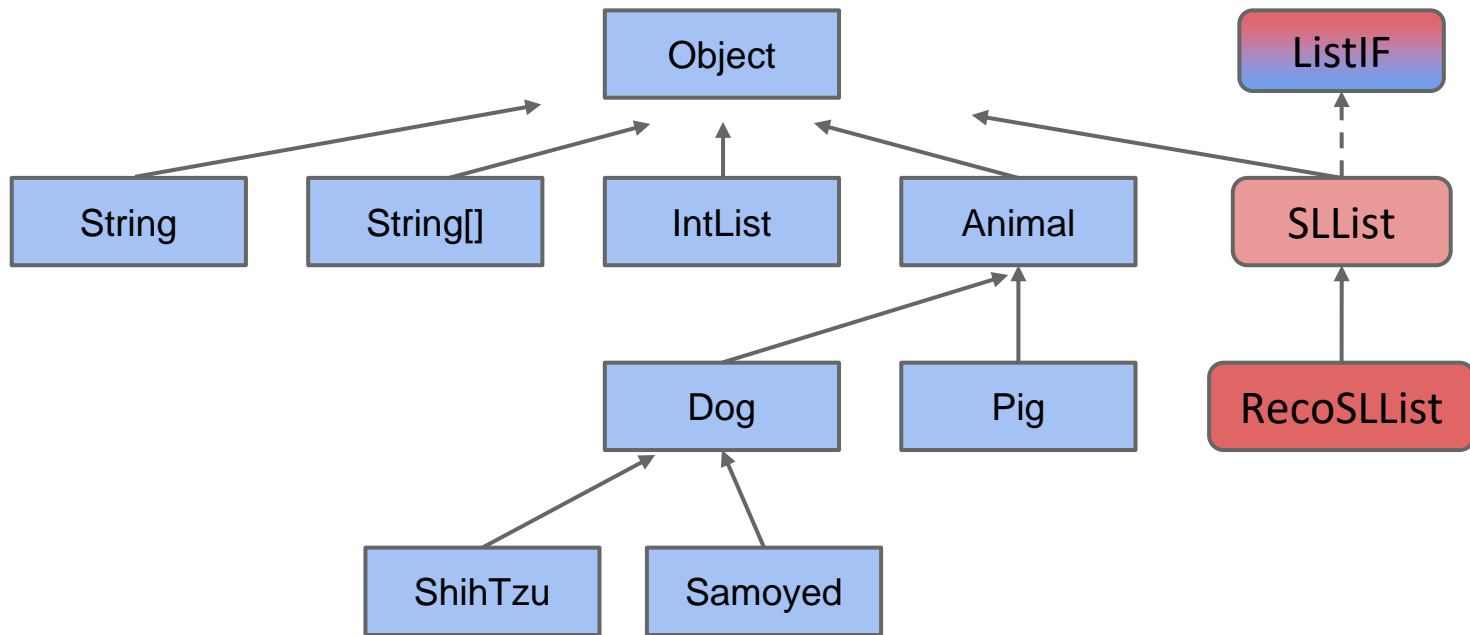
The Object Class (2)

- `Object` class is a concrete class (as opposed to abstract class)
    - it has implementations, which means your class actually inherits:

Modifier and Type	Method	Description
protected `Object`	`clone()`	Creates and returns a copy of this object.
boolean	`equals` `(Object obj)`	Indicates whether some other object is "equal to" this one.
protected void	`finalize()`	**Deprecated.** The finalization mechanism is inherently problematic.
`Class<?>`	`getClass()`	Returns the runtime class of this `Object`.
int	`hashCode()`	Returns a hash code value for the object.
void	`notify()`	Wakes up a single thread that is waiting on this object's monitor.
void	`notifyAll()`	Wakes up all threads that are waiting on this object's monitor.
`String`	`toString()`	Returns a string representation of the object.
void	`wait()`	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*.
void	`wait` `(long timeout)`	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.
void	`wait` `(long timeout, int nanos)`	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.

we will be using

Another Example (1)

- Suppose `Dog` has a method `bark()` that is overridden by `ShihTzu`

    ○ For each `bark()`, is there an error, or if not, which `bark()` is called?

```
Object obj1 = new ShihTzu("Baobei", 5);

ShihTzu s1 = (ShihTzu) obj1;
s1.bark();

Dog d1 = (Dog) obj1;
d1.bark();

((Dog) obj1).bark();

Object obj2 = (Dog) obj1;
obj2.bark();
```

Another Example (2)

- Suppose `Dog` has a method `bark()` that is overridden by `ShihTzu`
  - For each `bark()`, is there an error, or if not, which `bark()` is called?

```
Object obj1 = new ShihTzu("Baobei", 5);

ShihTzu s1 = (ShihTzu) obj1;
s1.bark();

Dog d1 = (Dog) obj1;
d1.bark();

((Dog) obj1).bark();

Object obj2 = (Dog) obj1;
obj2.bark();
```

ShihTzu is an Object, so this is allowed

Object is cast into a ShihTzu, and then assigned to a ShihTzu -> allowed

ShihTzu.bark() is called

Another Example (3)

- Suppose `Dog` has a method `bark()` that is overridden by `ShihTzu`
  - For each `bark()`, is there an error, or if not, which `bark()` is called?

```
Object obj1 = new ShihTzu("Baobei", 5);

ShihTzu s1 = (ShihTzu) obj1;
s1.bark();

Dog d1 = (Dog) obj1;
d1.bark();

((Dog) obj1).bark();

Object obj2 = (Dog) obj1;
obj2.bark();
```

Object is cast into a Dog, and
then assigned to a Dog -> allowed

1. Check if static type Dog has bark: yes
2. Dynamic type ShihTzu override it: yes
-> ShihTzu.bark() is called

Another Example (4)

- Suppose `Dog` has a method `bark()` that is overridden by `ShihTzu`
  - For each `bark()`, is there an error, or if not, which `bark()` is called?

```
Object obj1 = new ShihTzu("Baobei", 5);

ShihTzu s1 = (ShihTzu) obj1;
s1.bark();

Dog d1 = (Dog) obj1;
d1.bark();

((Dog) obj1).bark();

Object obj2 = (Dog) obj1;
obj2.bark();
```

The same as the previous line
1. Check if static type Dog has bark: yes
2. Dynamic type ShihTzu override it: yes
-> ShihTzu.bark() is called

Casting does **not** have lasting effect,
it does **not** change the static type of obj1,
it only enforce a static type within the (...)

Another Example (5)

- Suppose `Dog` has a method `bark()` that is overridden by `ShihTzu`
  - For each `bark()`, is there an error, or if not, which `bark()` is called?

```
Object obj1 = new ShihTzu("Baobei", 5);

ShihTzu s1 = (ShihTzu) obj1;
s1.bark();

Dog d1 = (Dog) obj1;
d1.bark();

((Dog) obj1).bark();

Object obj2 = (Dog) obj1;
obj2.bark();
```

because casting, the static type of right hand side is Dog, next, is a Dog an Object? yes -> allowed

Another Example (6)

- Suppose `Dog` has a method `bark()` that is overridden by `ShihTzu`
  - For each `bark()`, is there an error, or if not, which `bark()` is called?

```
Object obj1 = new ShihTzu("Baobei", 5);

ShihTzu s1 = (ShihTzu) obj1;
s1.bark();

Dog d1 = (Dog) obj1;
d1.bark();

((Dog) obj1).bark();

Object obj2 = (Dog) obj1;
obj2.bark();
```

the static type of obj2 is Object
does an Object define a bark()
method? no
-> compile error

Max Function

- Suppose we want to write a method `max()` that returns the max of **any** array, regardless of the type

Max Function

- Suppose we want to write a method `max()` that returns the max of **any** array, regardless of the type

```java
public static Object max(Object[] items) {
    int maxIndex = 0;
    for (int i = 0; i < items.length; i += 1) {
        if (items[i] > items[maxIndex]) {
            maxIndex= i;
        }
    }
    return items[maxIndex];
}
```

say your friend wrote this code.
there is a bug in the code.
can you find where the bug is?

```java
public static void main(String[] args) {
    Dog[] dogs = {new Dog("Baobei", 5), new Dog("Jiaozi", 7),
                    new Dog("Xiaxue", 20)};
    Dog maxDog = (Dog) StatLib.max(dogs);
    maxDog.bark();
}
```

In-Class Quiz 2

- Suppose we want to write a method `max()` that returns the max of **any** array, regardless of the type

```java
public static Object max(Object[] items) {
    int maxIndex = 0;
    for (int i = 0; i < items.length; i += 1) {
        if (items[i] > items[maxIndex]) {
            maxIndex= i;
        }
    }
    return items[maxIndex];
}
```

say your friend wrote this code.
there is a bug in the code.
can you find where the bug is?

1

2

```java
public static void main(String[] args) {
    Dog[] dogs = {new Dog("Baobei", 5), new Dog("Jiaozi", 7),
                  new Dog("Xiaxue", 20)};
    Dog maxDog = (Dog) StatLib.max(dogs);
    maxDog.bark();
}
```

3

Max Function Bug (1)

- Suppose we want to write a method `max()` that returns the max of **any** array, regardless of the type

```java
public static Object max(Object[] items) {
    int maxIndex = 0;
    for (int i = 0; i < items.length; i += 1) {
        if (items[i] > items[maxIndex]) {
            maxIndex= i;
        }
    }
    return items[maxIndex];
}
```

we pass an array of Dog here.
because a Dog is an Object,
an array of Dog is also an
array of Object.
no bug here.

```java
public static void main(String[] args) {
    Dog[] dogs = {new Dog("Baobei", 5), new Dog("Jiaozi", 7),
                  new Dog("Xiaxue", 20)};
    Dog maxDog = (Dog) StatLib.max(dogs);
    maxDog.bark();
}
```

this is a good use of casting.
no bug here.

Max Function Bug (2)

- Suppose we want to write a method `max()` that returns the max of *any* array, regardless of the type

```java
public static Object max(Object[] items) {
    int maxIndex = 0;
    for (int i = 0; i < items.length; i += 1) {
        if (items[i] > items[maxIndex]) {
            maxIndex= i;
        }
    }
    return items[maxIndex];
}
```

the > operator does not work with arbitrary object types. this **is** a bug.

```java
public static void main(String[] args) {
    Dog[] dogs = {new Dog("Baobei", 5), new Dog("Jiaozi", 7),
                  new Dog("Xiaxue", 20)};
    Dog maxDog = (Dog) StatLib.max(dogs);
    maxDog.bark();
}
```

what should we do if we want to compare and find max dog?

Max Dog

- In order to make the `max()` method work on your Dog class:

  - Step 1: make your Dog class comparable by implementing the `Comparable<T>` interface

    ```
    public interface Comparable<T> {
        public int compareTo(T other);
    }
    ```

  - Step 2: make your max method call compareTo to compare the Comparable objects in an array of Comparables

Step 1: Comparable Dog

- To make your `Dog` class comparable:

  - Declare the class implementing the `Comparable<T>` interface

    ```java
    public class Dog implements Comparable<Dog> {
    ```

  - Implement the method `compareTo`

    ```java
    @Override
    public int compareTo(Dog other) {

        if (this.weight < other.weight) {
            return -1;
        } else if (this.weight == other.weight) {
            return 0;
        }
        return 1;
    }
    ```

    we compare the dogs based on their weights
    return -1 if smaller than other dog
    return 0 if the same
    return 1 if larger

Step 1: Comparable Dog 2

- To make your `Dog` class comparable:

  - Declare the class implementing the `Comparable<T>` interface

    ```java
    public class Dog implements Comparable<Dog> {
    ```

  - Implement the method `compareTo`

    ```java
    @Override
    public int compareTo(Dog other) {


        return this.weight - other.weight;


    }
    ```

    alternatively,
    return negative if smaller than other dog
    return 0 if the same
    return positive if larger

Step 2: Max Comparables

- Make your `max` method call `compareTo` to compare the `Comparable` objects in an array of `Comparables`

```java
public static Comparable max(Comparable[] items) {
    int maxIndex = 0;
    int cmp;
    for (int i = 0; i < items.length; i++) {

        cmp = items[i].compareTo(items[maxIndex]);

        if (cmp > 0) {
            maxIndex = i;
        }
    }
    return items[maxIndex];
}
```

compareTo will return a positive number if items[maxIndex] is larger

now you can run your max method on any Comparable objects!
this is called **Generic Methods**

Equality: Introduction (1)

- Now we will discuss about equality

- In the first part of the lecture, we have developed a rigorous notion of data abstraction by creating types that are characterized by their operations, not by their representation

  - For an abstract data type, the abstraction function explains how to *interpret a concrete representation value as a value of the abstract type*, and we saw how the choice of abstraction function determines how to write the code implementing each of the ADT's operations

- In this part we turn to how we define the notion of **equality of values in a data type**: the abstraction function will give us a way to cleanly define the equality operation on an ADT

Equality: Introduction (2)

- In the *physical world*, every object is distinct — at some level, even two snowflakes are different, even if the distinction is just the position they occupy in space

    - This isn't strictly true of all subatomic particles, but true enough of large objects like snowflakes, baseballs and people

    - So two physical objects are never truly "equal" to each other; they only have degrees of similarity

- In the world of *human language*, however, and in the world of *mathematical concepts*, you can have multiple names for the same thing

    - so it's natural to ask when two expressions represent the same thing: 1+2, √9, and 3 are alternative expressions for the same ideal mathematical value

Three Ways to Regard Equality (1)

Formally, we can regard equality in several ways

- **Using an abstraction function**. Recall that an abstraction function AF: R → A maps concrete instances of a data type to their corresponding abstract values
    - To use AF as a definition for equality, we would say that a equals b if and only if AF(a)=AF(b)
    - For example, when we use strings that allows duplicates and at the same time require that the characters of that strings to be sorted, to represent sets of characters
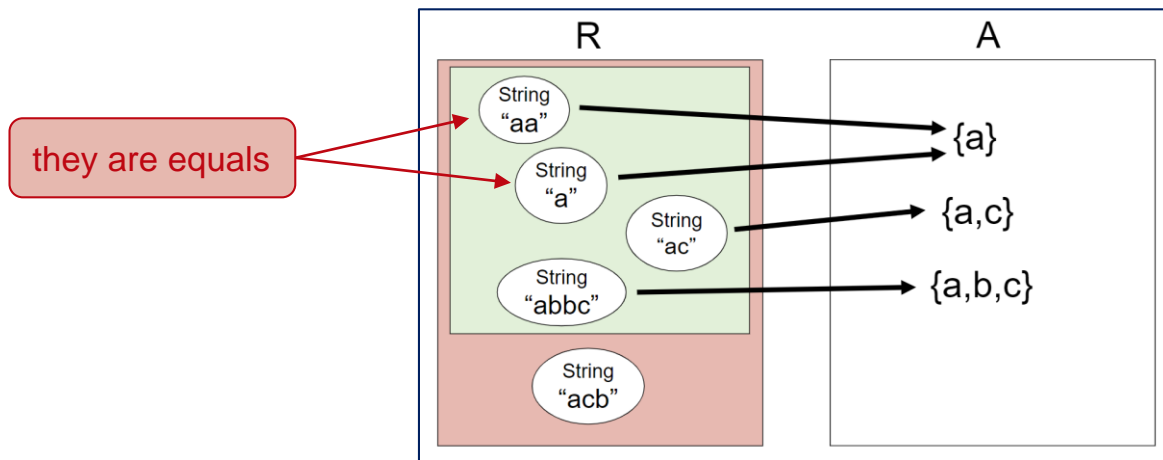
Three Ways to Regard Equality (2)

- **Using a relation**. An equivalence is a relation $E \subseteq T \times T$ that is:
    - reflexive: $E(t,t) \; \forall \, t \in T$
    - symmetric: $E(t,u) \Rightarrow E(u,t)$
    - transitive: $E(t,u) \wedge E(u,v) \Rightarrow E(t,v)$

  To use E as a definition for equality, we would say that a equals b if and only if $E(a,b)$

  These two notions are ***equivalent***:
    - An equivalence relation induces an abstraction function (the relation partitions T, so f maps each element to its partition class)
    - The relation induced by an abstraction function is an equivalence relation (check for yourself that the three properties hold)

Three Ways to Regard Equality (2)

A third way we can talk about the equality between abstract values is in terms of what an outsider (a client) can observe about them

- **Using observation**. We can say that two objects are equal when they *cannot be distinguished by observation* — **every operation** we can apply **produces the same result for both objects** Consider the set expressions {1,2} and {2,1} — using the observer operations available for sets, cardinality |...| and membership ∈, these expressions are indistinguishable:

  - |{1,2}| = 2 and |{2,1}| = 2

  - 1 ∈ {1,2} is true, and 1 ∈ {2,1} is true

  - 2 ∈ {1,2} is true, and 2 ∈ {2,1} is true

  - 3 ∈ {1,2} is false, and 3 ∈ {2,1} is false

In terms of abstract data types, *observation* means calling operations on the objects

So **two objects are equal** if and only if **they cannot be distinguished by calling any operations** of the abstract data type

In-Class Quiz 3

- Consider the following rep for an abstract data type:

```java
/** Immutable type representing a set of letters, ignoring case */
class LetterSet {
    private String s;
    // Abstraction function:
    //     AF(s) = the set of the letters that are found in s
    //             (ignoring non-letters and alphabetic case)
    // Rep invariant:
    //     true

    /**
     * Make a LetterSet consisting of the letters found in chars
     * (ignoring alphabetic case and non-letters).
     */
    public LetterSet(String chars) {
        s = chars;
    }

    ... // observer and producer operations
}
```

- Using the *abstraction function* definition of equality for LetterSet, which of the following should be considered equal to new LetterSet("abc")?

- ❑ new LetterSet("aBc")
- ❑ new LetterSet("")
- ❑ new LetterSet("bbbbbc")
- ❑ new LetterSet("1a2b3c")

== vs equals() (1)

- Like many languages, Java has **two** different operations for testing equality, with different semantics

- The **== operator** compares *references*
  - more precisely, it tests **referential equality**
  - two references are == if they point to *the same storage* in memory
  - in terms of the snapshot diagrams we've been drawing, two references are == if their arrows point to the same object bubble

- The `equals()` operation compares *object contents*
  - in other words, **object equality**, in the sense that we've been talking about in this lecture
  - the equals operation has to be defined appropriately for every abstract data type

== vs equals()   (2)

- For comparison, here are the equality operators in several languages:

	referential equality	object equality
Java	==	equals()
Objective C	==	isEqual:
C#	==	Equals()
Python	is	==
Javascript	==	n/a

== vs equals() (3)

- Note that == unfortunately *flips its meaning* between Java and Python!

  - Don't let that confuse you: == in Java just tests reference identity, it *doesn't* compare object contents

- As programmers in any of these languages, we can't change the meaning of the referential equality operator

  - In Java, == always means *referential equality*

  - But when we define a new data type, it's our responsibility to decide what object equality means for values of the data type, and implement the `equals()` operation appropriately

equals in ARSet.contains

- In fact, you have encountered this issue on your `ARSet.contains` method

```java
public boolean contains(T item) {
    if (item == null) {
        return false;
    }
    for (int i = 0; i < size; i++) {
        if (items[i].equals(item)) {
            return true;
        }
    }
    return false;
}
```

use **equals**, instead of **==**
your set may contain objects with the same content, but located at different address in memory

as a consequence, you need to have a special case if you allow to add and store null, because equals called on null will cause runtime NullPointerException

An Example

- Here's a simple example of an immutable ADT we will be using in discussing equality:

```java
public class Duration {
    private final int mins;
    private final int secs;
    // rep invariant:
    //     mins >= 0, secs >= 0
    // abstraction function:
    //     represents a span of time of mins minutes and secs seconds
    /** Make a duration lasting for m minutes and s seconds. */
    public Duration(int m, int s) {
        mins = m; secs = s;
    }
    /** @return length of this duration in seconds */
    public int getLength() {
        return mins*60 + secs;
    }
}
```

Equality of Immutable Types (1)

- As previously mentioned, the `equals()` method is defined by `Object`, and its default implementation looks like this:

```
public class Object {
    ...
    public boolean equals(Object that) {
        return this == that;
    }
}
```

- In other words, the **default meaning** of `equals()` is **the same** as referential equality!

- For immutable data types, this is almost always wrong!

  - So you have to ***override*** the `equals()` method, replacing it with your own implementation
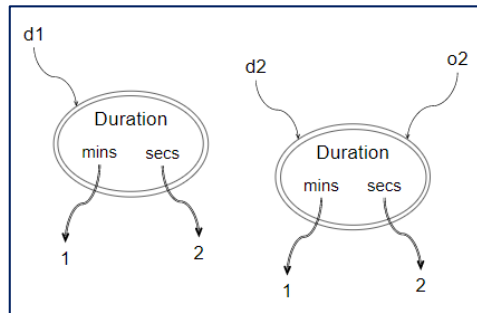
Equality of Immutable Types (2)

- Here's our first try for `Duration`:

```java
public class Duration {

    ...
    // Problematic definition of equals()
    public boolean equals(Duration that) {
        return this.getLength() == that.getLength();
    }

}
```

- There's a subtle problem here. Why doesn't this work? Let's try this code:

```java
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 2);
Object o2 = d2;
d1.equals(d2) → true
d1.equals(o2) → false
```

Equality of Immutable Types (3)

- You'll see that even though `d2` and `o2` end up referring to the very same object in memory, you still get different results for them from `equals()`

- What's going on?

- It turns out that `Duration` has ***overloaded*** the `equals()` method, because the method signature was **not** identical to `Object`'s !

    - we actually have **two** `equals()` methods in `Duration`: an implicit `equals(Object)` inherited from `Object`, and the new `equals(Duration)`

Equality of Immutable Types (4)

```java
public class Duration extends Object {

    // explicit method that we declared:

    public boolean equals (Duration that) {

        return this.getLength() == that.getLength();

    }

    // implicit method inherited from Object:

    public boolean equals (Object that) {

        return this == that;

    }

}
```

equals is overloaded !

Equality of Immutable Types (5)

- Recall that the Java compiler selects between overloaded operations using the compile-time type of the parameters

  - for example, when you use the / operator, the compiler *chooses* either integer division or floating-point division based on whether the arguments are `ints` or `doubles`

- The same compile-time selection happens here

  - If we pass an `Object` reference, as in `d1.equals(o2)`, we end up calling the `equals(Object)` implementation

  - If we pass a `Duration` reference, as in `d1.equals(d2)`, we end up calling the `equals(Duration)` version

  - This happens even though `o2` and `d2` both point to the same object at runtime!

  - Equality has become inconsistent

Equality of Immutable Types (6)

- It's easy to make a mistake in the method signature, and overload a method when you meant to override it

- Recall last week, that this is such a common error that Java has a language feature, the annotation `@Override`, which you should use whenever your intention is to override a method in your superclass

  - With this annotation, the Java compiler will check that a method with the same signature actually exists in the superclass, and give you a compiler error if you've made a mistake in the signature

Equality of Immutable Types (7)

- So here's the right way to implement `Duration`'s `equals()` method:

```java
@Override
public boolean equals (Object that) {
    if (that == null) return false;
    if (!(that instanceof Duration)) return false;
    Duration thatDuration = (Duration) that;
    return this.getLength() == thatDuration.getLength();

}
```

- This fixes the problem:

```java
Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 2);
Object o2 = d2;
d1.equals(d2) → true
d1.equals(o2) → true
```

instanceof

- `instanceof` tests whether an object is an instance of a particular type

- Using `instanceof` is dynamic type checking, not the static type checking we prefer

- In our course and most good Java programming: `instanceof` is *disallowed* anywhere **except for implementing equals**

  - this prohibition also includes other ways of inspecting objects' runtime types

  - for example, `getClass()` is also *disallowed*

Optimization

- To optimize the code, no further comparison needed if they are the exact same object:

```java
@Override
public boolean equals (Object that) {
    if (this == that) return true;
    if (that == null) return false;
    if (!(that instanceof Duration)) return false;
    Duration thatDuration = (Duration) that;
    return this.getLength() == thatDuration.getLength();
}
```

located at the same place in memory

The Object Contract

- The specification of the `Object` class is so important that it is often referred to as the **Object Contract**
  - The contract can be found in the method specifications for the `Object` class
  - Here we will focus on the contract for `equals`

- When you override the equals method, you must adhere to its general contract, that states:
  - equals must define **an equivalence relation** — that is, a relation that is reflexive, symmetric, and transitive
  - equals must be **consistent**: repeated calls to the method must yield the same result provided no information used in equals comparisons on the object is modified
  - for a **non-null** reference x, `x.equals(null)` should return false
  - **hashCode** must produce the same result for two objects that are deemed equal by the `equals` method

will talk about this next week

Breaking the Equivalence Relation (1)

- Let's start with the equivalence relation

- We have to make sure that the definition of equality implemented by `equals()` is actually an equivalence relation as defined earlier: reflexive, symmetric, and transitive

- If it isn't, then operations that depend on equality (like sets, searching) will behave erratically and unpredictably

- You don't want to program with a data type in which sometimes a equals b, but b doesn't equal a

  - subtle and painful bugs will result

Breaking the Equivalence Relation (2)

- Here's an example of how an innocent attempt to make equality more flexible can go wrong

- Suppose we wanted to allow for a tolerance in comparing `Duration` objects, because different computers may have slightly unsynchronized clocks:

```java
private static final int CLOCK_SKEW = 5; // seconds
@Override
public boolean equals (Object thatObject) {
    if (!(thatObject instanceof Duration)) return false;
    Duration thatDuration = (Duration) thatObject;
    return Math.abs(this.getLength() - thatDuration.getLength()) <= CLOCK_SKEW;
}
```

- This will violate the equivalence relation

Equality of Mutable Types (1)

- We've been focusing on equality of immutable objects; what about mutable objects?

- Equality must still be an equivalence relation, as required by the `Object` contract

- We also want equality to respect the abstraction function and respect operations

    - But with mutable objects, there is a new possibility: by calling a mutator on one of the objects before doing the observation, we may change its state and thus create an observable difference between the two objects

Equality of Mutable Types (2)

So let's refine our definition and allow for two notions of equality based on observation:

- **Observational equality** means that two references *cannot be distinguished now*, in the current state of the program
  - A client can try to distinguish them *only by calling operations that don't change the state of either object* (i.e. only observers and producers, not mutators) and *comparing the results* of those operations
  - This tests whether the two references "look" the same for the current state of the object

- **Behavioral equality** means that two references cannot be distinguished now or in the future, *even if a mutator is called to change the state of one object but not the other*
  - This tests whether the two references will "behave" the same, in this and all future states

often, we implement behavioral equality with referential equality ==

Equality of Mutable Types (3)

- For immutable objects, observational and behavioral equality are *identical*
  - because there aren't any mutator methods that can change the state of the objects

- For *mutable objects*, it can be very tempting to use observational equality as the design choice, but we *should use behavioral equality*
  - Java uses observational equality for most of its mutable data types, in fact
  - If two distinct `List` objects contain the same sequence of elements, then `equals()` reports that they are equal
  - But the presence of mutators unfortunately leads to subtle bugs, because it means that equality isn't consistent over time
  - Two objects that are observationally equal at one moment in the program may stop being equal after a mutation

- We will continue talking about this next week

> so, for mutable data structure,
> we sometimes use observational,
> and sometimes we use behavioral equality

Equality of Mutable Types (4)

- Consider these three lists of integers:

```
List<Integer> listA = List.of(1, 2, 3);
List<Integer> listB = new ArrayList<>(listA);
List<Integer> listC = listB;
```

- If we want to explore whether the `List` values referred to by `listA`, `listB`, and `listC` are equal by **observational equality**, what could we do?

- Which objects are equal by **observational equality**?

Equality of Mutable Types (5)

- Consider these three lists of integers:

```
List<Integer> listA = List.of(1, 2, 3);
List<Integer> listB = new ArrayList<>(listA);
List<Integer> listC = listB;
```

- If we want to explore whether the `List` values referred to by `listA`, `listB`, and `listC` are equal by **behavioral equality**, what could we do?

- Which objects are equal by **behavioral equality**?

Thank you for your attention !

- In this lecture, you have learned:
  - about invariant that is always true of an ADT object instance for its lifetime
  - to use abstraction function and to avoid representation exposure
  - to use casting carefully
  - about using dynamic method selection
  - to make a class or a data structure support comparability
  - to differentiate between reference equality and object equality
  - to implement equality correctly for mutable and immutable types

- Please continue to Lecture Quiz 11 and Lab 11:
  - to do Lab Exercise 11.1 - 11.2, and
  - to do Exercise 11.1 - 11.2