CSE 12:
Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Fourteen
24 Aug 2011

More on generics.

Collections to hold data of type `T`

- Up to now we have discussed generics in its simplest usage -- store data of an arbitrary type `T` in a container.
    - This worked fine for lists/arrays/stacks/queues, in which we ignore any *order relations* among the elements.
- Sometimes, however, the type `T` cannot be "just any old `Object`" -- type `T` must sometimes *satisfy some conditions*.

Constraints on **`T`**

- An example of this is the **`HeapImpl12`** class you are building for P4.
  - The elements must all be **`Comparable`** -- the heap implementation needs to be able to call **`compareTo(o)`** on every element stored in the tree.
  - If we place no restrictions on **`T`**, then the Java compiler cannot guarantee that an arbitrary element of the **`_nodeArray`** will actually be **`Comparable`**.

Constraints on T

- Suppose we add three objects to a heap:

```
heap = new Heap12<Object>();
heap.add("Michael");  // OK: String is Comparable
heap.add("Bolton");  // OK: String is Comparable
heap.add(new Object()); // Not OK: Object not Comparable
```

- Internally, the **HeapImpl12** class will need to call **compareTo** on all objects to implement **bubbleUp** and **trickleDown**, e.g.:

```
if (_nodeArray[idx1].compareTo(_nodeArray[idx2]) < 0) {
   ...
}
```

But if **idx1** refers to the **Object** we added, this method will fail because **Object** does not implement the **Comparable** interface.

Bounds on type parameters

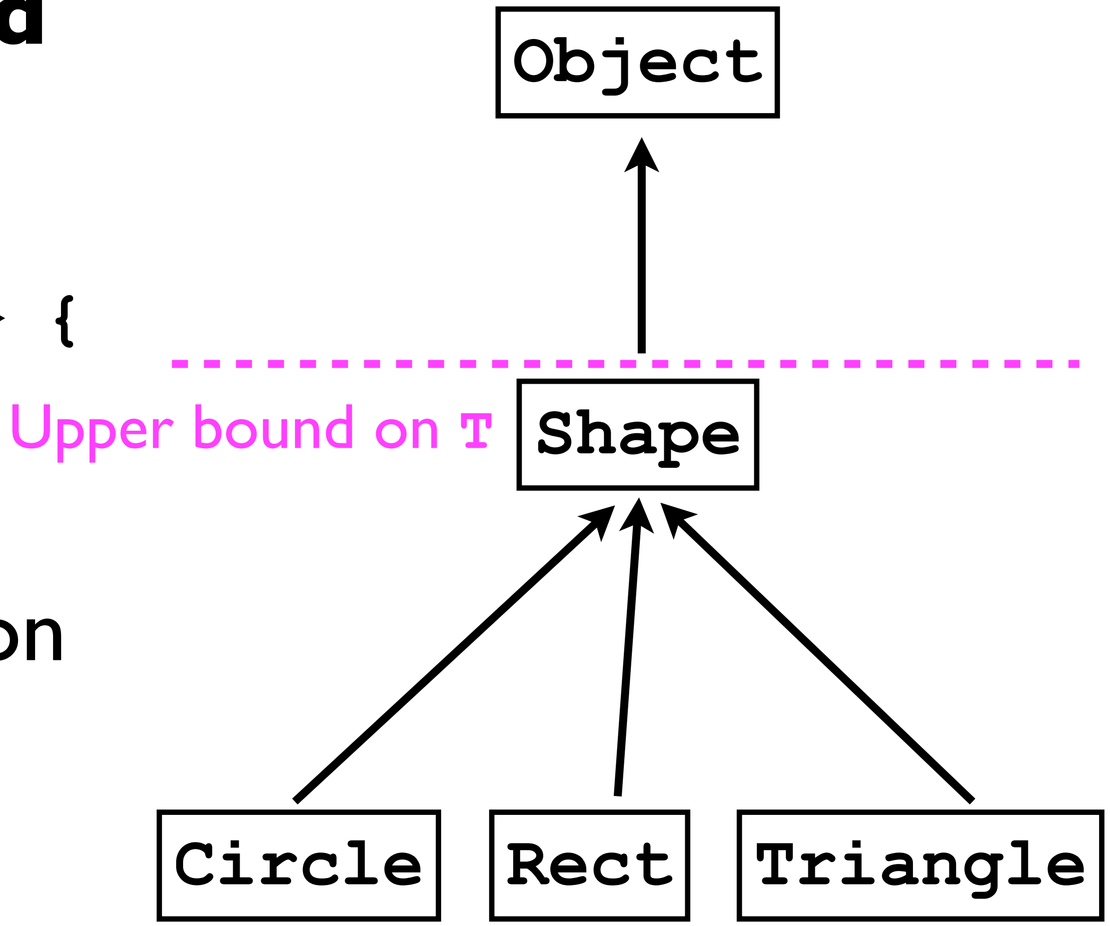
- What we want is a way of *enforcing* that the type parameter **`T`** allowed by the **`HeapImpl12`** class -- as well as the **`Heap12`** interface itself -- be of type **`Comparable`**.
- Java generics facilitates these constraints on **`T`** by supporting **bounds** on type parameters.
- Suppose, when implementing a generic class with type parameter **`T`**, we want to *ensure* that **`T`** must be *some sub-class* of a class **`A`**.
    - *Example*: we want to implement a container for **`Shape`** objects -- we don't care what *particular* kind of **`Shape`**s they are, so long as they all *inherit from* the **`Shape`** class.

Bounds on type parameters

- To implement a generic class with the guarantee that type parameter **`T`** is a **`Shape`**, we can use an **upper bound** on **`T`**:

```
class MyContainer<T extends Shape> {
   ...
}
```

- Here, **`Shape`** is the **upper bound** on type parameter **`T`**.
  - **`MyContainer`** can only be instantiated when **`T`** is **`Shape`**, or *any sub-class of* **`Shape`**.



Bounds on type parameters

- Given this upper bound on `T`, the Java compiler will enforce that `T` be of type `Shape`:

```
MyContainer<Shape> container1 =
  new MyContainer<Shape>();  // OK

MyContainer<Circle> container2 =
  new MyContainer<Circle>();  // OK

MyContainer<Object> container4 =
  new MyContainer<Object>();  // Not OK
```

```
Compiler error message:
  type parameter java.lang.Object is not within its bound
  MyContainer<Object> container4 = new MyContainer<Object>();
```

```
MyContainer<Student> container3 =
  new MyContainer<Student>();  // Not OK
```

Bounds on type parameters

- We can also require that type **`T`** *implement* some interface.
  - For example, a **`HeapImpl12`** should only store elements that are all **`Comparable`**.
- Java generics gives us this power:

```
class HeapImpl12<T extends Comparable> implements Heap12<T> {
   ...
}
```

- The "**`extends Comparable`**" enforces that any **`T`** we pass in as the type parameter *must* be of type **`Comparable`**.
  - Since **`Comparable`** is an *interface*, this means that type **`T`** must *implement* the interface **`Comparable`** (even though we use the word "**`extends`**").

Bounds on type parameters

- With this restriction on `T` in place, we can no longer instantiate a `HeapImpl12` with a type parameter `T` that does not implement `Comparable`:

```
// String and Integer are both Comparable
HeapImpl12<String> heap1 = new HeapImpl12<String>();  // OK
HeapImpl12<Integer> heap2 = new HeapImpl12<Integer>(); // OK

// Next line won’t compile because Object is not Comparable
HeapImpl12<Object> heap3 = new HeapImpl12<Object>();
```

- The Java *compiler* will prevent us from instantiating a heap with a non-`Comparable` type.

- We may also wish to define the *interface* `Heap12` to accept only those types `T` that implement `Comparable`:

```
interface Heap12<T extends Comparable> {
  ...
}
```

Bounds on type parameters

- In the previous example, `Comparable` was the upper bound of `T`.
- The `Comparable` interface takes a type parameter of its own.

  ```
  interface Comparable<U> {
    int compareTo (U o);
  }
  ```

  (In the previous example, we used the `Comparable` interface in “compatibility mode”, where we did not specify `U`).

- The type parameter `U` specifies what kinds of objects `o` we should be able to compare to.

Bounds on type parameters

- By offering **bounds** on type parameters, Java also gives us the power to define what kinds of objects `U` we can `compareTo`, *in terms of the type* `T` *we've already defined*.
- Example:

```
class HeapImpl12<T extends Comparable<T>> ... {
  ...
}
```

- Here, we require that whatever type `T` the `HeapImpl12` is instantiated with, it *must* be `Comparable` to *other objects of type* `T`.

Bounds on type parameters

- Consider the following example:

  ```
  class B { }
  class A implements Comparable<B> {
    int compareTo (B o) {
      return 0;
    }
  }
  ```

- Given the definitions above, an object of type A can *only* be compared to objects of type `B`.

  ```
  final A a = new A();
  final B b = new B();
  final int result = a.compareTo(b);  // OK
  ```

  - We *cannot* compare `a` to another object of type `A`!

Bounds on type parameters

- Given our definition of HeapImpl12,

  ```
  class HeapImpl12<T extends Comparable<T>> ... {
    ..
  }
  ```

  if we try to instantiate a `HeapImpl12` with `A` as the type parameter...

  ```
  HeapImpl12<A> heap = new HeapImpl12<A>();
  ```

  ... the compiler will complain:

  ```
   type parameter A is not within its bound
    HeapImpl12<A> h = new HeapImpl12<A>();
  ```

- This error occurs because, even though `A` is `Comparable` to *something* (`B`), it is not `Comparable<A>`.

Bounds on type parameters

- On the other hand,
  - **`String`** implements **`Comparable<String>`**
  - **`Integer`** implements **`Comparable<Integer>`**
- Both **`String`** and **`Integer`** would be accepted as type parameters for **`HeapImpl12`**:

```
HeapImpl12<String> h1 = new HeapImpl12<String>();
HeapImpl12<Integer> h2 = new HeapImpl12<Integer>();
```

Both are OK

Bounds on type parameters

- While useful, our current definition of **`HeapImpl12`** is a bit *overly restrictive*.
- Consider a hierarchy of **`Shape`** classes:

```
class Shape implements Comparable<Shape> {
   int compareTo (Shape o) { ... }
}
class Rectangle extends Shape {
  ...
}
```

- The **`Rectangle`** class inherits the **`compareTo (Shape o)`** method from its parent **`Shape`** class.

Bounds on type parameters

- However, `Rectangle` does not offer a method `compareTo (Rectangle o)` designed specifically for other `Rectangle` objects.

- Hence, the `Rectangle` class *could not be used* as the type parameter `T` when instantiating a `HeapImpl12`:

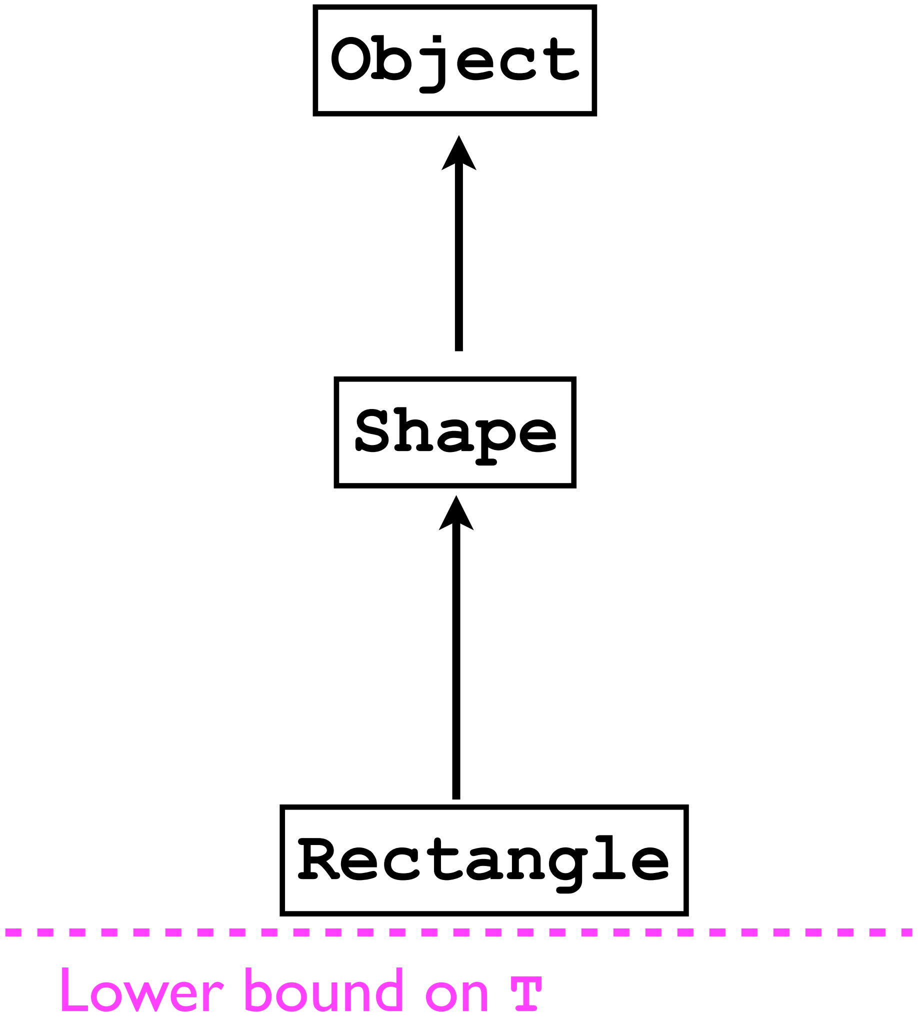
```
class HeapImpl12<T extends Comparable<T>> ...
```

  - *Reason*: Even though `Rectangle` is `Comparable` to other `Shape` objects, it is not `Comparable<Rectangle>`.

    - I.e., `Rectangle` offers no `int compareTo (Rectangle o)` method.

Lower bounds on types

- What we need is a way of expressing that type parameter `T` may be `Comparable` with class `T`, *or any super-class of* `T`.
    - E.g., we want to allow `HeapImpl12` to store `Rectangle` objects:
        - `Rectangles` are all `Comparable` with `Shape`, where `Shape` is a *super-class* of `Rectangle`.
- To solve this problem, Java offers **lower bounds** on type parameters.



Lower bounds on types

- For example, we can allow the **`HeapImpl12`** class to accept any type **`T`** so long as **`T`** is **`Comparable`** to class **`T`**, or any super-class of **`T`**.

```
class HeapImpl12<T extends Comparable<? super T>> ... {
   ...
}
```

- The wildcard type **`?`** indicates:
  - "We don't care which type **`T`** is **`Comparable`** to, so long as it's **`Comparable`** to some super-class of **`T`** (or **`T`** itself)."
    - The keyword **`super`** indicates the **lower bound** of the type parameter.

Lower bounds on types

- Given this revised definition of `HeapImpl12`, we can now instantiate a heap of `Rectangle` objects:

```
HeapImpl12<Rectangle> heap =
  new HeapImpl12<Rectangle>();  // OK
```