INFO1113 / COMP9003
Object-Oriented Programming

Lecture 11

THE UNIVERSITY OF SYDNEY

Contents developed by Tyson Thomas

Acknowledgement of Country

I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.

I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.

Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Topics: Part A

- Wildcards
- **extends** with Wildcard
- **super** with Wildcard

Wildcards

We are going back to using generics but we will be exploring the wildcards a little further.

Unlike arrays, where we are able to assign types to a variable of an inherited type, Generic types cannot be assigned to a type that specifies a lower bound.

In Java

arrays are covariant

generics are invariant

Can we assign inherited types with generics?

Wildcards

The answer is **No.**

However, we are able to **read** super types using wildcards and **write** to a list knowing its lower bound.

Wildcards

```
class Person {
   String name;
   public Person(String name) {
      this.name = name;
   }
   public String toString() { return name; }
}

class Employee extends Person {
   int salary;
   public Employee(String name, int salary) {
      super(name);
      this.salary = salary;
   }
   public String toString() { return "[" + name + "," + salary + "]"; }
}

class Customer extends Person {
   int customerID;
   public Customer(String name, int customerID) {
      super(name);
      this.customerID = customerID;
   }
   public String toString() { return "[" + name + "," + customerID + "]"; }
}

public class GenericsWildcard {

   public static void readPeople(List<Person> people) {
      for(Person p : people) {
         System.out.println(p);
      }
   }

   public static void main(String[] args) {
      List<Employee> employees = new ArrayList<Employee>();
      employees.add(new Employee("Jeff", 76559));
      employees.add(new Employee("Alice", 27584));
      employees.add(new Employee("Kelly", 4332));
      readPeople(employees);
   }
}
```

Creating an **ArrayList** that contains Employee objects. We can see that Employee is a type of **Person**.

Wildcards

```
class Person {
   String name;
   public Person(String name) {
      this.name = name;
   }
   public String toString() { return name; }
}

class Employee extends Person {
   int salary;
   public Employee(String name, int salary) {
      super(name);
      this.salary = salary;
   }
   public String toString() { return "[" + name + "," + salary + "]"; }
}

class Customer extends Person {
   int customerID;
   public Customer(String name, int customerID) {
      super(name);
      this.customerID = customerID;
   }
   public String toString() { return "[" + name + "," + customerID + "]"; }
}

public class GenericsWildcard {

   public static void readPeople(List<Person> people) {
      for(Person p : people) {
         System.out.println(p);
      }
   }

   public static void main(String[] args) {
      List<Employee> employees = new ArrayList<Employee>();
      employees.add(new Employee("Jeff", 76559));
      employees.add(new Employee("Alice", 27584));
      employees.add(new Employee("Kelly", 4332));
      readPeople(employees);
   }
}
```

Given the List requires person the compiler will refuse to accept a list that contains a subtype.

Wildcards

Given some class that utilises generics, we are able to specify a wildcard by using **?** symbol. This will allow the many different types to be associated with the container.

Syntax:

Type<?> variable;

Type<? super LowerBound> variable;

Type<? extends UpperBound> variable;

Wildcards

Given some class that utilises generics, we are able to specify a wildcard by using **?** symbol. This will allow the many different types to be associated with the container.

Syntax:

Type<?> variable;

Type<? super LowerBound> variable;

Type<? extends UpperBound> variable;

Example:

```
List<?> list;
List<? extends Person> people;
List<? super Employee> employees;
```

Wildcards

Given some class that utilises generics, we are able to specify a wildcard by using **?** symbol. This will allow the many different types to be associated with the container.

Syntax:

Type<?> variable;

Type<? super LowerBound> variable;

Type<? extends UpperBound> variable;

Example:

```
List<?> list;
List<? extends Person> people;
List<? super Employee> employees;
```

Given an upper bound, any type we retrieve from this collection can be treated as the **upper bound**.

Wildcards

Given some class that utilises generics, we are able to specify a wildcard by using **?** symbol. This will allow the many different types to be associated with the container.

Syntax:

Type<?> variable;

Type<? super LowerBound> variable;

Type<? extends UpperBound> variable;

Example:

```
List<?> list;
List<? extends Person> people;
List<? super Employee> employees;
```

Given a collection which can contain its lower bound, we can only assume the types we read from this will be of Object but we can add **Employee** objects to it.

Wildcards

Given some class that utilises generics, we are able to specify a wildcard by using **?** symbol. This will allow the many different types to be associated with the container.

Syntax:

Type<?> variable;

Type<? super LowerBound> variable;

Type<? extends UpperBound> variable;

Example:

```
List<?> list;
List<? extends Person> people;
List<? super Employee> employees;
```

We specified no lower or upper bound, this is a wild card which we can only read from and treat only as an object

So how do we read all the employees?

Wildcards

Let's fix this

```
class Person {
   String name;
   public Person(String name) {
      this.name = name;
   }
   public String toString() { return name; }
}

class Employee extends Person {
   int salary;
   public Employee(String name, int salary) {
      super(name);
      this.salary = salary;
   }
   public String toString() { return "[" + name + "," + salary + "]"; }
}

public class GenericsWildcard {

   public static void readPeople(List<Person> people) {
      for(Person p : people) {
         System.out.println(p);
      }
   }

   public static void main(String[] args) {
      List<Employee> employees = new ArrayList<Employee>();
      employees.add(new Employee("Jeff", 76559));
      employees.add(new Employee("Alice", 27584));
      employees.add(new Employee("Kelly", 4332));
      readPeople(employees);
   }
}
```

```
class Customer extends Person {
   int customerID;
   public Customer(String name, int customerID) {
      super(name);
      this.customerID = customerID;
   }
   public String toString() { return "[" + name + "," + customerID + "]"; }
}
```

Wildcards

Let's fix this

```
class Person {
   String name;
   public Person(String name) {
      this.name = name;
   }
   public String toString() { return name; }
}

class Employee extends Person {
   int salary;
   public Employee(String name, int salary) {
      super(name);
      this.salary = salary;
   }
   public String toString() { return "[" + name + "," + salary + "]"; }
}

class Customer extends Person {
   int customerID;
   public Customer(String name, int customerID) {
      super(name);
      this.customerID = customerID;
   }
   public String toString() { return "[" + name + "," + customerID + "]"; }
}

public class GenericsWildcard {

   public static void readPeople(List<? extends Person> people) {
      for(Person p : people) {
         System.out.println(p);
      }
   }

   public static void main(String[] args) {
      List<Employee> employees = new ArrayList<Employee>();
      employees.add(new Employee("Jeff", 76559));
      employees.add(new Employee("Alice", 27584));
      employees.add(new Employee("Kelly", 4332));
      readPeople(employees);
   }
}
```

We have used a **wildcard** and provided an upper bound for the generic.

Wildcards

Let's fix this

```
class Person {
   String name;
   public Person(String name) {
      this.name = name;
   }
   public String toString() { return name; }
}

class Employee extends Person {
   int salary;
   public Employee(String name, int salary) {
      super(name);
      this.salary = salary;
   }
   public String toString() { return "[" + name + "," + salary + "]"; }
}

class Customer extends Person {
   int customerID;
   public Customer(String name, int customerID) {
      super(name);
      this.customerID = customerID;
   }
   public String toString() { return "[" + name + "," + customerID + "]"; }
}

public class GenericsWildcard {

   public static void readPeople(List<? extends Person> people) {
      for(Person p : people) {
         System.out.println(p);
      }
   }

   public static void main(String[] args) {
      List<Employee> employees = new ArrayList<Employee>();
      employees.add(new Employee("Jeff", 76559));
      employees.add(new Employee("Alice", 27584));
      employees.add(new Employee("Kelly", 4332));
      readPeople(employees);
   }
}
```

Since the upper bound has been specified we are able to treat all object within this list as the upper bound

Wildcards

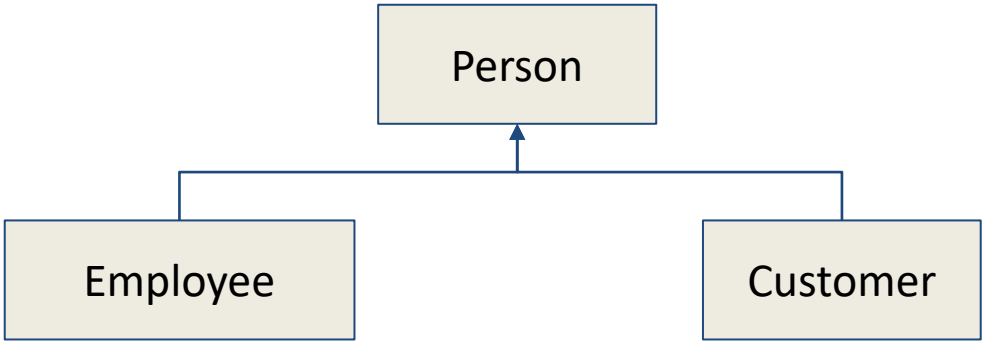
Let's fix this

```
class Person {
   String name;
   public Person(String name) {
      this.name = name;
   }
   public String toString() { return name; }
}

class Employee extends Person {
   int salary;
   public Employee(String name, int salary) {
      super(name);
      this.salary = salary;
   }
   public String toString() { return "[" + name + "," + salary + "]"; }
}

class Customer extends Person {
   int customerID;
   public Customer(String name, int customerID) {
      super(name);
      this.customerID = customerID;
   }
   public String toString() { return "[" + name + "," + customerID + "]"; }
}

public class GenericsWildcard {

   public static void readPeople(List<? extends Person> people) {
      for(Person p : people) {
         System.out.println(p);
      }
   }

   public static void main(String[] args) {
      List<Employee> employees = new ArrayList<Employee>();
      employees.add(new Employee("Jeff", 76559));
      employees.add(new Employee("Alice", 27584));
      employees.add(new Employee("Kelly", 4332));
      readPeople(employees);
   }
}
```

```
> java GenericsWildcard
[Jeff,76559]
[Alice,27584]
[Kelly,4332]
<Program End>
```

Wildcards

For the following inheritance hierarchy:



The following can be assigned to:

```
List<? extends Person> people = new ArrayList<Employee>();
List<? extends Person> people = new ArrayList<Customer>();
List<? extends Person> people = new ArrayList<Person>();
```

Wildcards

However, we couldn't add any objects to the list because we cannot assume what type the variable is bound to.

Otherwise we could add **Customer** to a list that contains **Employee** or **Person** objects to **Customer** lists.

Okay, so what about the super keyword?

Wildcards

```
class Media {
   String title;
   public Media(String title) { this.title = title; }
}

class Book extends Media {
   int pageCount;
   public Book(String name, int pageCount) {
      super(name);
      this.pageCount = pageCount;
   }
}
class Video extends Media {
   double duration;
   public Video(String name, double duration) {
      super(name);
      this.duration = duration;
   }
}

public class GenericWrite {
   public static void addBook(List<Media> media, Book b) {
      media.add(b);
   }
   public static void main(String[] args) {
      List<Media> m = new ArrayList<Media>();
      List<Book> b = new ArrayList<Book>();

      addBook(m, new Book("Pride and Prejudice", 432));
      addBook(b, new Book("War and Peace", 1225));
   }
}
```

Wildcards

```
class Media {
   String title;
   public Media(String title) { this.title = title; }
}

class Book extends Media {
   int pageCount;
   public Book(String name, int pageCount) {
      super(name);
      this.pageCount = pageCount;
   }
}
class Video extends Media {
   double duration;
   public Video(String name, double duration) {
      super(name);
      this.duration = duration;
   }
}

public class GenericWrite {
   public static void addBook(List<Media> media, Book b) {
      media.add(b);
   }
   public static void main(String[] args) {
      List<Media> m = new ArrayList<Media>();
      List<Book> b = new ArrayList<Book>();

      addBook(m, new Book("Pride and Prejudice", 432));
      addBook(b, new Book("War and Peace", 1225));
   }
}
```

Let's consider the case where we have two collection types, one contains Media and the other contains Books.

Wildcards

```
class Media {
   String title;
   public Media(String title) { this.title = title; }
}

class Book extends Media {
   int pageCount;
   public Book(String name, int pageCount) {
      super(name);
      this.pageCount = pageCount;
   }
}
class Video extends Media {
   double duration;
   public Video(String name, double duration) {
      super(name);
      this.duration = duration;
   }
}

public class GenericWrite {
   public static void addBook(List<Media> media, Book b) {
      media.add(b);
   }
   public static void main(String[] args) {
      List<Media> m = new ArrayList<Media>();
      List<Book> b = new ArrayList<Book>();

      addBook(m, new Book("Pride and Prejudice", 432));
      addBook(b, new Book("War and Peace", 1225));
   }
}
```

We want to add Book objects to both lists

Wildcards

```
class Media {
   String title;
   public Media(String title) { this.title = title; }
}

class Book extends Media {
   int pageCount;
   public Book(String name, int pageCount) {
      super(name);
      this.pageCount = pageCount;
   }
}
class Video extends Media {
   double duration;
   public Video(String name, double duration) {
      super(name);
      this.duration = duration;
   }
}

public class GenericWrite {
   public static void addBook(List<Media> media, Book b) {
      media.add(b);
   }
   public static void main(String[] args) {
      List<Media> m = new ArrayList<Media>();
      List<Book> b = new ArrayList<Book>();

      addBook(m, new Book("Pride and Prejudice", 432));
      addBook(b, new Book("War and Peace", 1225));
   }
}
```

However, the **addBook** method only accepts a list with the type argument of **Media.**

Wildcards

```
class Media {
   String title;
   public Media(String title) { this.title = title; }
}

class Book extends Media {
   int pageCount;
   public Book(String name, int pageCount) {
      super(name);
      this.pageCount = pageCount;
   }
}
class Video extends Media {
   double duration;
   public Video(String name, double duration) {
      super(name);
      this.duration = duration;
   }
}

public class GenericWrite {
```

```
> Javac GenericWrite.java
error: incompatible types: List<Book> cannot be converted to List<Media>
      addBook(b, new Book("War and Peace", 1225));
              ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get
full output
1 error
```

We can specify a lower bound on the list

Wildcards

```
class Media {
   String title;
   public Media(String title) { this.title = title; }
}

class Book extends Media {
   int pageCount;
   public Book(String name, int pageCount) {
      super(name);
      this.pageCount = pageCount;
   }
}
class Video extends Media {
   double duration;
   public Video(String name, double duration) {
      super(name);
      this.duration = duration;
   }
}

public class GenericWrite {
   public static void addBook(List<? super Book> media, Book b) {
      media.add(b);
   }
   public static void main(String[] args) {
      List<Media> m = new ArrayList<Media>();
      List<Book> b = new ArrayList<Book>();

      addBook(m, new Book("Pride and Prejudice", 432));
      addBook(b, new Book("War and Peace", 1225));
   }
}
```

Wildcards

```
class Media {
   String title;
   public Media(String title) { this.title = title; }
}

class Book extends Media {
   int pageCount;
   public Book(String name, int pageCount) {
      super(name);
      this.pageCount = pageCount;
   }
}
class Video extends Media {
   double duration;
   public Video(String name, double duration) {
      super(name);
      this.duration = duration;
   }
}

public class GenericWrite {
   public static void addBook(List<? super Book> media, Book b) {
      media.add(b);
   }
   public static void main(String[] args) {
      List<Media> m = new ArrayList<Media>();
      List<Book> b = new ArrayList<Book>();

      addBook(m, new Book("Pride and Prejudice", 432));
      addBook(b, new Book("War and Peace", 1225));
   }
}
```

Given the list must be able to contain a **Book** we specify parameter to accept any list which can contain Book and its super types.

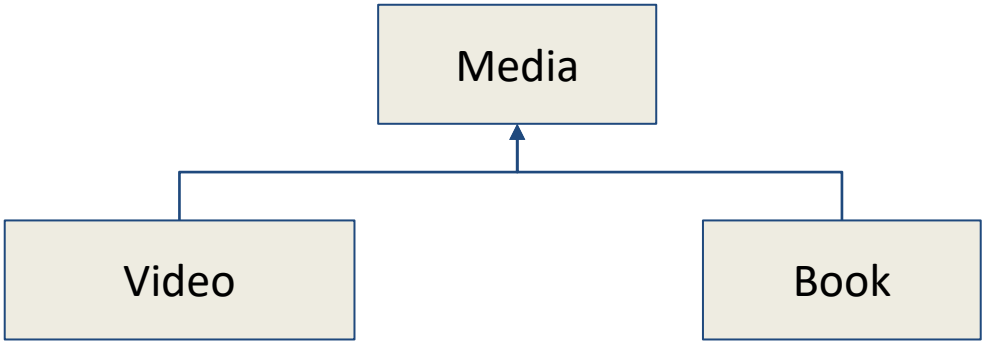
Wildcards

```
class Media {
   String title;
   public Media(String title) { this.title = title; }
}

class Book extends Media {
   int pageCount;
   public Book(String name, int pageCount) {
      super(name);
      this.pageCount = pageCount;
   }
}
class Video extends Media {
   double duration;
   public Video(String name, double duration) {
      super(name);
      this.duration = duration;
   }
}

public class GenericWrite {
   public static void addBook(List<? super Book> media, Book b) {
      media.add(b);
   }
   public static void main(String[] args) {
      List<Media> m = new ArrayList<Media>();
      List<Book> b = new ArrayList<Book>();

      addBook(m, new Book("Pride and Prejudice", 432));
      addBook(b, new Book("War and Peace", 1225));
   }
}
```

We can pass **List<Media>** and **List<Book>** to the **addBook** method without any compilation error.

Wildcards

Considering the inheritance heirarchy, we are able to specify the lower bound which can **add** the lower bound but nothing else.

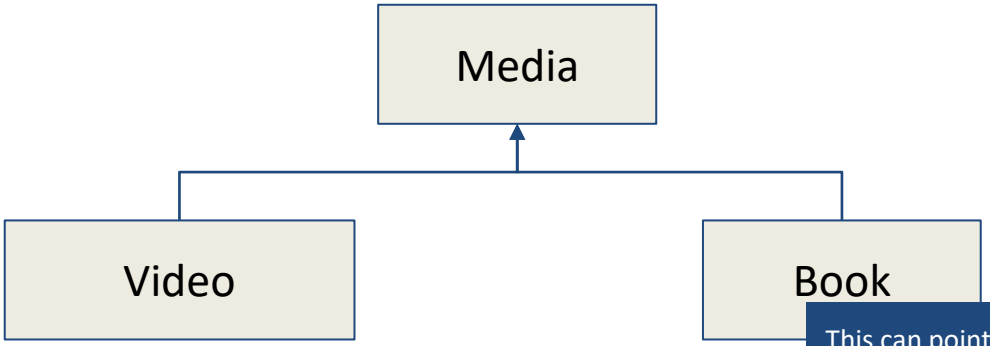


The following can be assigned to:

```
List<? super Book> books = new ArrayList<Book>();
List<? super Book> media = new ArrayList<Media>();
```

Wildcards

Considering the inheritance heirarchy, we are able to specify the lower bound which can **add** the lower bound but nothing else.



This can point to a Media list and therefore the list can contain **Media** types within it, however since it can also point to a **Book** list, we cannot add a type that isn't the lower bound.

The following can be assigned to:

```
List<? super Book> books = new ArrayList<Book>();
List<? super Book> media = new ArrayList<Media>();
```

Demo

Debugging at runtime

JDB

Logic errors are much harder to detect than syntax errors. Within Java, the compiler can display when we have incorrectly written a statement, but it cannot outline if there is an error in our logic, we just see incorrect output.

So, how do we break down what is going on?

JDB

The java debugger provides functionality to

- Inspect the current values of local variables
- Stop execution and step through the code
- Inspect object and class instances

This simple tool allows you to inspect the state of your program and understand where an error has occurred.

JDB

We can set up **breakpoints** that allows us to pause execution and inspect the current state of the program.

When the program is paused, we can inspect **local** variables, **dump** object information, simply **print** an object value or **set** a variable to a specific value.

Overriding values allows us to check scenarios which may not be easy to replicate.

JDB

We can set up **breakpoints** that allows us to pause execution and inspect the current state of the program.

We can specify a breakpoint using the commands **stop at**

Example: stop at Program:53

Once it is paused, we can call **locals** and inspect all local variables to the method

When the program is paused, we can inspect **local** variables, **dump** object information, simply **print** an object value or **set** a variable to a specific value.

Overriding values allows us to check scenarios which may not be easy to replicate.

Similar to **dump** command but for specific variable, typically calling toString().

For any heavy control flow within the program, we can set the variable's value.

Similar to locals but for extracting information from an **object**. For methods that manipulate properties of an object we can inspect all properties.

How can we debug our programs?

JDB

So, we will need to ensure your program is compiled with debugging symbols.

```
> javac -g MyProgram.java
```

You can use jdb with a program compiled without debugging symbols but without this information we cannot easily inspect variables.

JDB

Once compiled, we can start a JDB session and inspect our program.

```
> jdb MyProgram
```

This starts a session, waiting for the user to set up necessary checks and run the program.

JDB

Once ready, we can set up breakpoints and then inset the program

```
> jdb MyProgram
Initializing jdb ...
> stop at MyProgram:32
Deferring breakpoint MyProgram:32.
It will be set after the class is loaded.
> run
run MyProgram
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint MyProgram:32

Breakpoint hit: "thread=main", MyProgram.main(), line=32 bci=6
32              op.execute("My String!");

main[1]
```

JDB

Once ready, we can set up breakpoints and then inset the program

```
> jdb MyProgram
Initializing jdb ...
> stop at MyProgram:32
Deferring breakpoint MyProgram:32.
It will be set after the class is loaded.
> run
run MyProgram
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint MyProgram:32

Breakpoint hit: "thread=main", MyProgram.main(), line=32 bci=6
32              op.execute("My String!");

main[1]
```

Setting a breakpoint on this line

JDB

Once ready, we can set up breakpoints and then inset the program

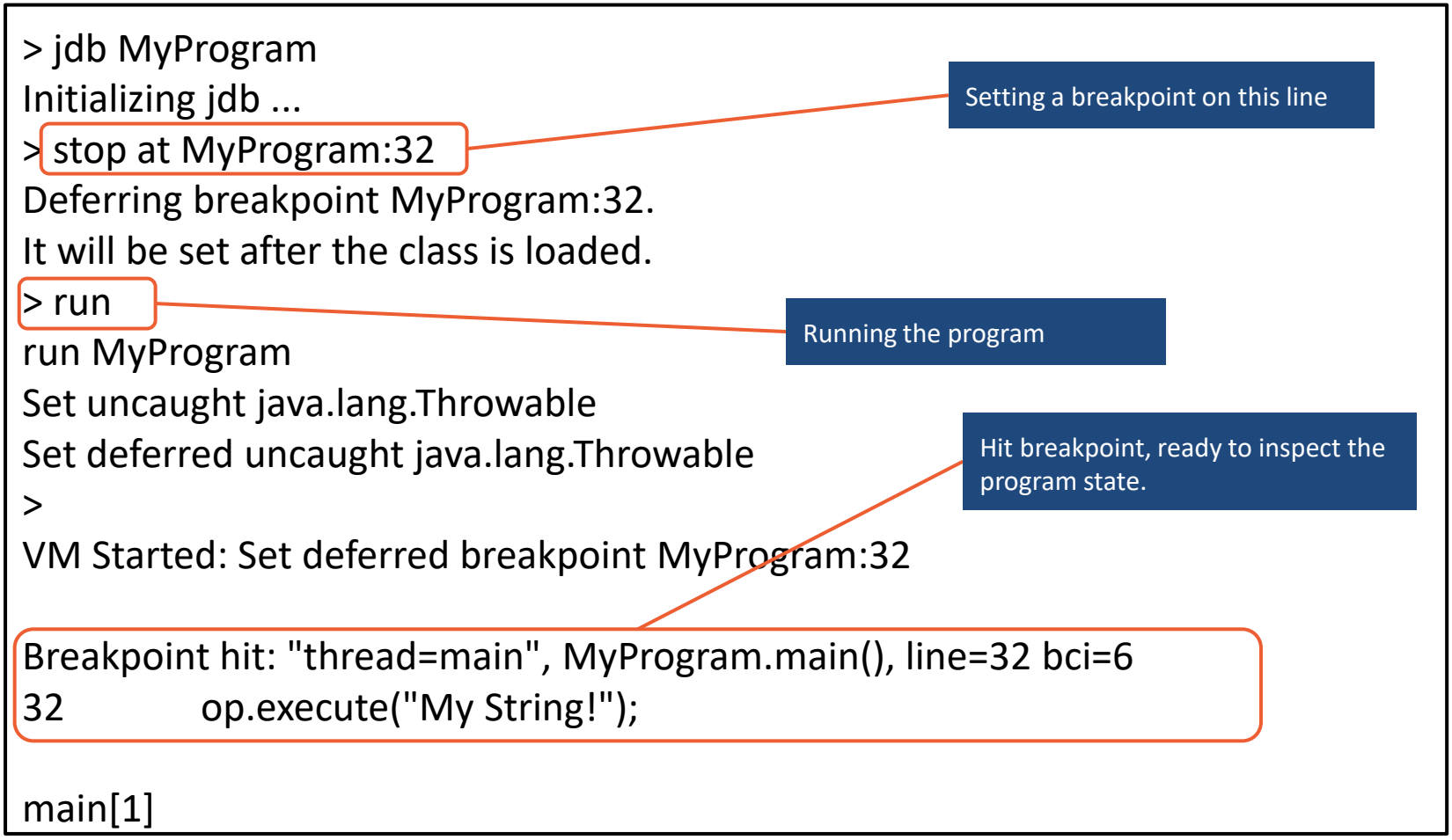
```
> jdb MyProgram
Initializing jdb ...
> stop at MyProgram:32
Deferring breakpoint MyProgram:32.
It will be set after the class is loaded.
> run
run MyProgram
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint MyProgram:32

Breakpoint hit: "thread=main", MyProgram.main(), line=32 bci=6
32              op.execute("My String!");

main[1]
```

Setting a breakpoint on this line

Running the program

JDB

Once ready, we can set up breakpoints and then inset the program



Let’s debug a program

JDB

Applications typically run for as long as they can (Webservers) so allow for functionality that allows us to inspect a currently running application gives this the opportunity to debug live programs.

This kind of interactions is what allows the programmer to debug programs requiring user input.

JDB

When running your program, we can set the virtual machine to allow debug and open a port for JDB to connect to.

```
> java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n MyProgram
```

Open up another window and use the following command. JDB will connect to the open session and interact with this.

```
> jdb –attach 8000
```

JDB

When running your program, we can set the virtual machine to allow debug and open a port for JDB to connect to.

```
> java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n MyProgram
```

Open up another window and use the following command. JDB will connect to the open session and interact with this.

```
> jdb –attach 8000
```

We have specified the socket the jdwp server is listening on and the port jdb will connect to.

Demo

See you next time!

THE UNIVERSITY OF SYDNEY