Java – why so popular (so quickly)

- Code looks like C (and C++) – familiar for many existing programmers
  - Object-oriented without complexities of C++
- Killer API (application programmers interface)
  - Built-in networking features
  - Graphical user interface (GUI) objects
  - Threads, media support, …
- Is free!
- Java virtual machine – JVM – "Write once, run anywhere."

A simple Java program

- Java “programs” – actually classes (types of objects)
- A first java application: class Hello
  1. Create file called `Hello.java`
  2. Compile – `javac Hello.java`
     (creates bytecode file named `Hello.class` if successful)
  3. Execute – `java Hello`
     (invokes JVM)

What is a Java application?

- Answer: A class with a `main` method

  e.g., `public static void main(String[] args){ }`

- Huh?

  `public` – can be invoked from another package

  `static` – same for all instances of this class

  `void` – does not return anything

  `main` – the method's name

  `(String[] args)` – parameter list (an array of Strings)

  `{ }` – block delimiters {method definition is inside}

Special characters & comments

- Escape sequences – all start with `\`
  - e.g., `\n` – newline, and `\t` – tab
  - Also `\"` – double quotes, and `\'` – single quote
  - `\\` – back slash itself, and more (see text p. 23)
  - Play with Hello.java – to see effects
- 3 types of comments:

```
// for single line or end-of-line comment

/* for comment that may
   span lines */

/** Javadoc comment (upcoming topic) */
```

Java has 8 primitive data types

- 7 are “number” types
  - 5 of the number types are *integral* types:
    - `int` – most fundamental; 4, -123, 9587123 are `int`
    - `long` – for longer integers (>2,147,483,647)
    - `short`, `byte` – save space for shorter integers
    - `char` – to represent characters; ‘A’, ’a’, ‘\n’
  - Other 2 number types are *floating point* types:
    - `double` – most fundamental; 0.4, -123.3, 95.
    - `float` – save space for less precision
- 8th type is `boolean`: to represent `true` or `false`
- Every other data type in Java is an object type

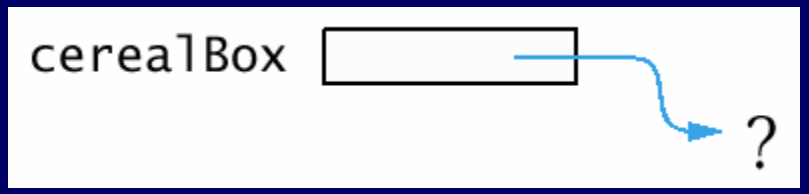
Objects

- An object is a thing or a concept
  - Often a model of a real-world thing or concept
- Probably contains both *data* and *methods* – i.e., a software object can *know* stuff, and *do* stuff
- Easy to create and use *(e.g., MoveTester.java)*:
  1. Declare reference – `Rectangle box;`
  2. Create the object, and assign it to the reference –

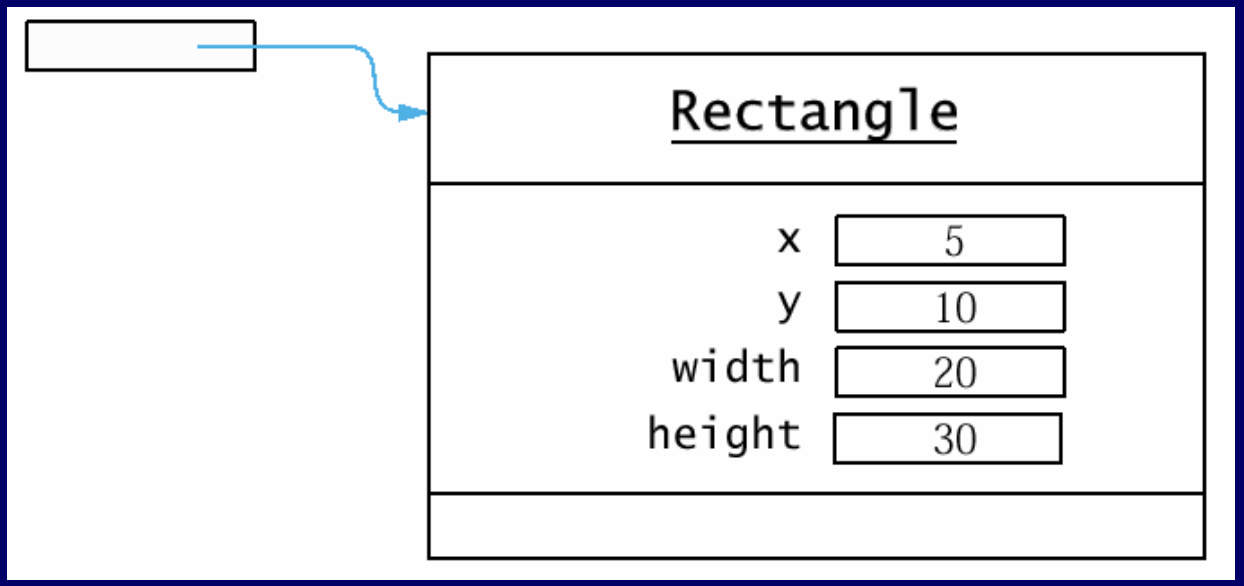
     ```
     box = new Rectangle(5, 10, 20, 30);
     ```
  3. Invoke its methods – `box.translate(15, 25);`

Objects vs. object references

A reference can "point" to nothing (`null`).



It must point to an actual object to be useful.



Classes

- Technically, an object is an instance of a class
- Classes define an object's *interface*
  - These are the *public* methods and data that other types of objects can access directly
  - e.g., Rectangle's translate() method
- Class definitions also contain the *implementation*
  - The *private* members and *internal details* of methods
  - e.g., x, y coordinates of Rectangle *should be* private data
  - e.g., how the translate method actually works to change these coordinates is unimportant to clients of the class

Bank account example

- Software design effort identified the need for objects that represent bank accounts
- Why *objects*, not just numbers?
  - Because bank accounts are more complex
    - Need a way to store a balance – data
    - But also need ways to deposit and withdraw money, and report the current balance – methods
- Idea is that other software objects will:
  - Create `new BankAccount` objects
  - Use the objects' methods to solve problems
- But first, must write `class BankAccount`

Class definition I: define the interface

```
public class BankAccount {
   public void deposit(double amount) {}
   public void withdraw(double amount) {}
   public double getBalance() {}
}
```

- This is all that programmers of other classes have to know: the public interface
  - They can start working independently – how methods are implemented doesn't matter
- Also the time to document the interface – add javadoc comments
  - More about javadoc comments later in course

Class definition II: define the data

- i.e., what objects of this class will "know"
- Variables declared outside any method
  - Includes *instance* variables
    - Can store different values for each instance (see text fig. 4, p. 39)
  - May also include static (a.k.a. "class") variables
- Tip: make instance variables private
  - e.g., `private double balance;`
  - Other classes can't directly access or alter
    - e.g., `harrysChecking.balance = -1000;` // error

Class definition III: implement the methods

- Often manipulate the data in some way

```
public void deposit(double amount) {
   balance = balance + amount;
}
public void withdraw(double amount) {
   balance = balance - amount;
}
```

- Other times provide a copy of the data

```
public double getBalance() {
   return balance;
}
```

Defining constructors

- A default constructor is always defined
  - e.g., `new BankAccount();` // no parameters
  - Initializes instance variables to default values:
    - Primitive number type values are set to `0`
    - `boolean` values are set to `false`
    - Object references are set to `null`
- Often want to "overload" the constructor
  - e.g., `public BankAccount(double initialBalance) { balance = initialBalance; }`
  - Name is same as classname, and there is no return type
- See `BankAccount.java` and `BankAccountTester.java`